Hugbúnaðarverkefni 1 / Software Project 1

8. Design Models

HBV501G – Fall 2018

Matthias Book

HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRUVÍSINDASVIÐ
IÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD

In-Class Quiz Prep

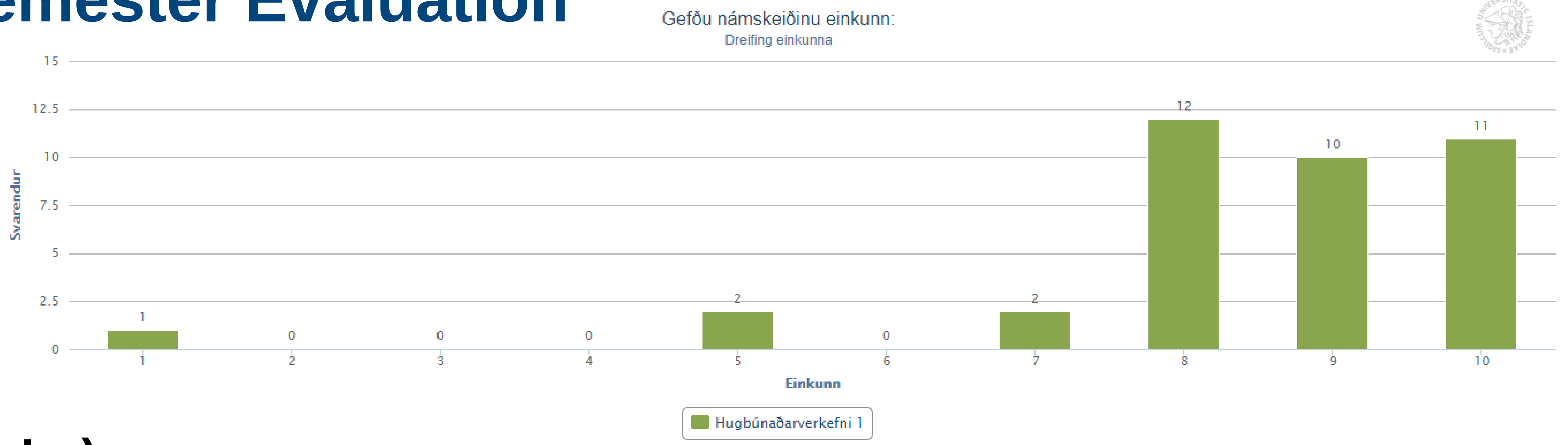
- Please prepare a scrap of paper with the following information:

- ID: _____@hi.is Date: ________
- a) ____________ e) ____________
- b) ____________ f) ____________
- c) ____________ g) ____________
- d) ____________ h) ____________

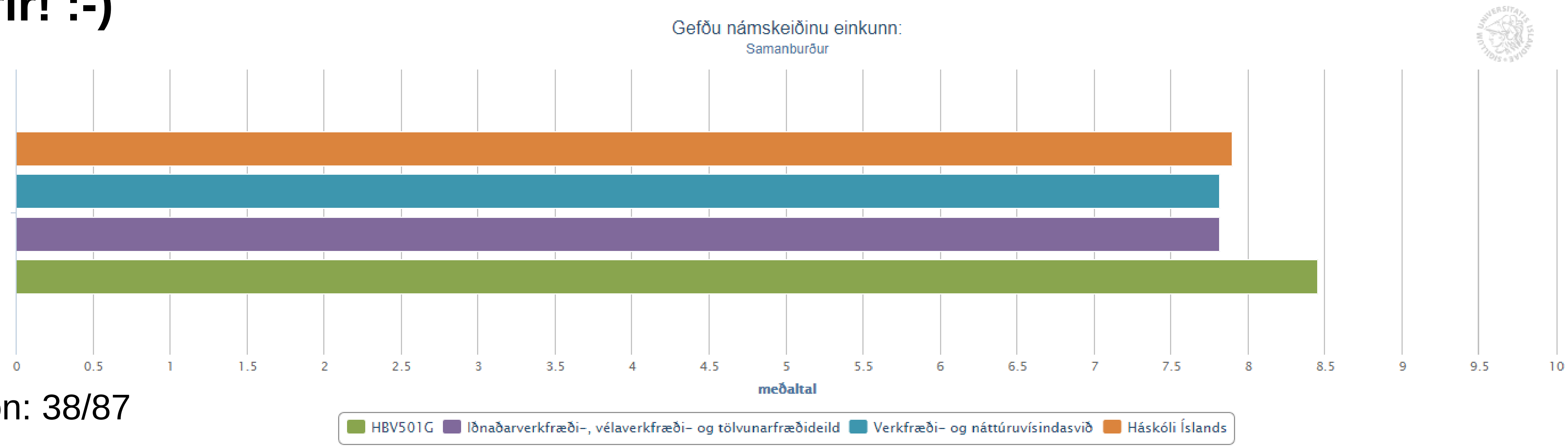
- During class, I'll show you questions that you can answer very briefly
  - No elaboration necessary
- Hand in your scrap at the end of class

- All questions in a quiz weigh same
- All quizzes (ca. 10 throughout semester) have the same weight
  - Your worst 2 quizzes will be disregarded
- Overall quiz grade counts as optional question worth 7.5% on final exam

Mid-Semester Evaluation



Takk fyrir! :-)



Participation: 38/87

Mid-Semester Evaluation – Open Feedback

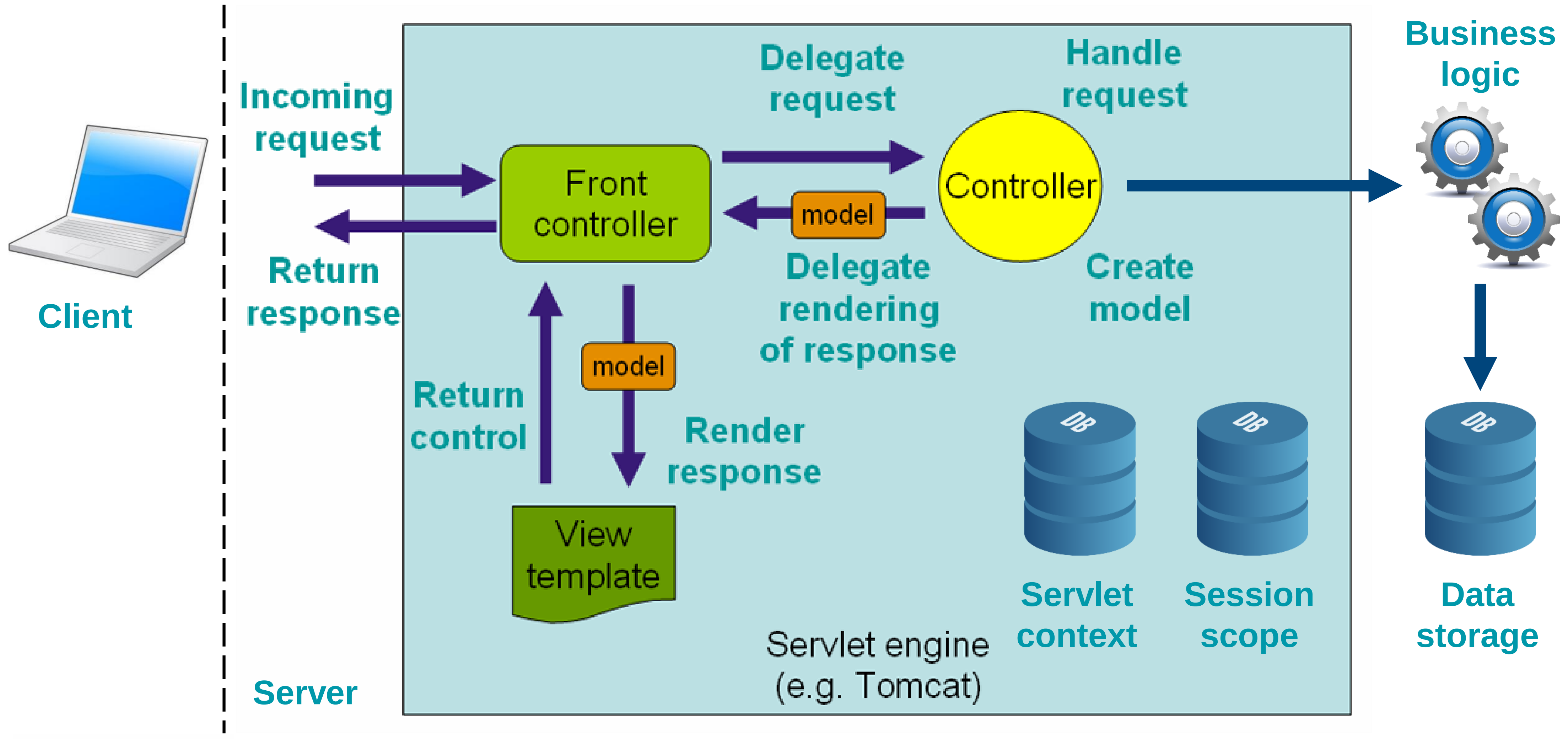
Things people like about the course

- Well-organized course
- Great teaching team
- Great to have free choice of project

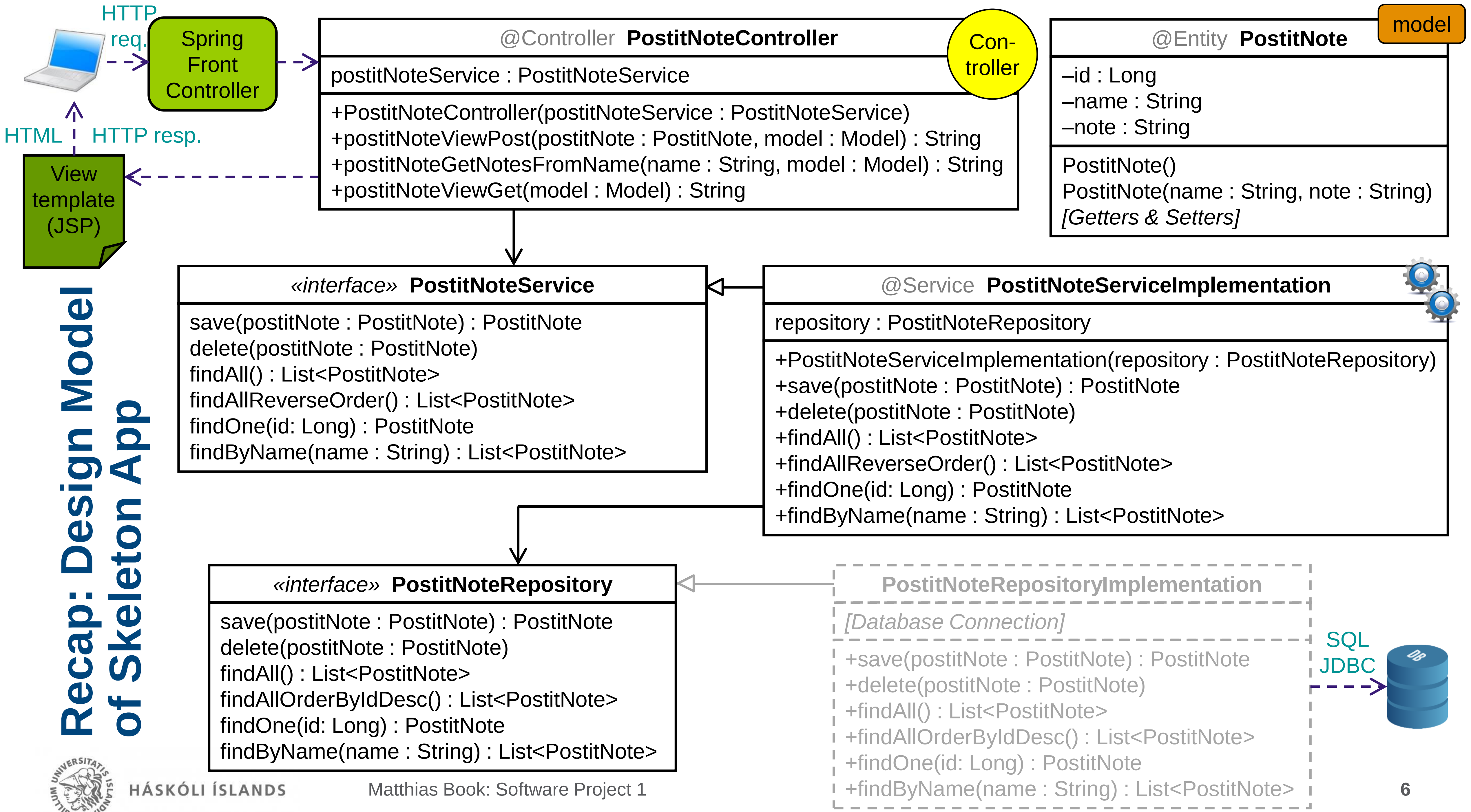
Things people would like to see improved

- Sometimes hard to stay focused in monotonous lectures
- More feedback in consultations
- Give more ideas for projects

Recap: Spring Web MVC Framework



Recap: Design Model of Skeleton App

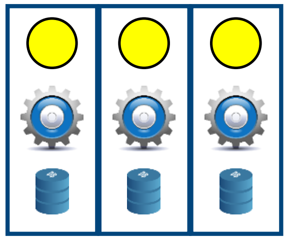

HTTP req.
Spring Front Controller
HTML
HTTP resp.
View template (JSP)
@Controller PostitNoteController
Con-troller
postitNoteService : PostitNoteService
+PostitNoteController(postitNoteService : PostitNoteService)
+postitNoteViewPost(postitNote : PostitNote, model : Model) : String
+postitNoteGetNotesFromName(name : String, model : Model) : String
+postitNoteViewGet(model : Model) : String
model
@Entity PostitNote
–id : Long
–name : String
–note : String
PostitNote()
PostitNote(name : String, note : String)
[Getters & Setters]
«interface» PostitNoteService
save(postitNote : PostitNote) : PostitNote
delete(postitNote : PostitNote)
findAll() : List<PostitNote>
findAllReverseOrder() : List<PostitNote>
findOne(id: Long) : PostitNote
findByName(name : String) : List<PostitNote>
@Service PostitNoteServiceImplementation
repository : PostitNoteRepository
+PostitNoteServiceImplementation(repository : PostitNoteRepository)
+save(postitNote : PostitNote) : PostitNote
+delete(postitNote : PostitNote)
+findAll() : List<PostitNote>
+findAllReverseOrder() : List<PostitNote>
+findOne(id: Long) : PostitNote
+findByName(name : String) : List<PostitNote>
«interface» PostitNoteRepository
save(postitNote : PostitNote) : PostitNote
delete(postitNote : PostitNote)
findAll() : List<PostitNote>
findAllOrderByIdDesc() : List<PostitNote>
findOne(id: Long) : PostitNote
findByName(name : String) : List<PostitNote>
PostitNoteRepositoryImplementation
[Database Connection]
+save(postitNote : PostitNote) : PostitNote
+delete(postitNote : PostitNote)
+findAll() : List<PostitNote>
+findAllOrderByIdDesc() : List<PostitNote>
+findOne(id: Long) : PostitNote
+findByName(name : String) : List<PostitNote>
SQL
JDBC
DB

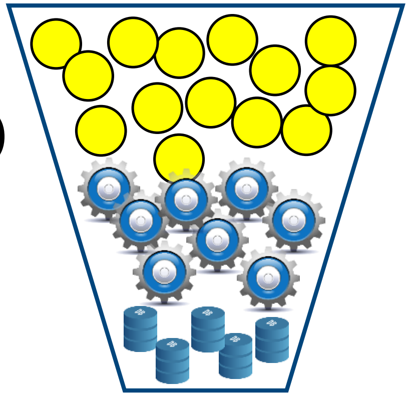
Component Responsibilities in Server-Side MVC Pattern

- **Responsibilities of a Controller**
  - Encapsulate Web specifics
    - Transform HTTP request parameters into domain-specific data structures
    - Call appropriate service methods (application logic)
    - Transform application logic outcomes into MVC model data
    - Determine the next view to be displayed, and return control to view template to produce response
    - Or return result data to client as a JSON string in a HTTP response (in a RESTController)
- **Responsibilities of a Service**
  - Encapsulate the technology-independent application logic
    - Implement the algorithms, decisions, data transformations etc. that fulfill your application domain requirements / business processes
    - Work with entities in object-oriented fashion, unaware of Web or DB specifics
- **Responsibilities of a Repository**
  - Ensure that entities are being stored persistently and retrieved from storage
    - Perform object-relational mapping
    - Take care of DB connection handling and other technicalities
    - Execute database operations
    - Transform query results into object-oriented Entities

Collaboration of Layers in More Complex Apps

- There's no necessity for a 1:1:1 relationship between Controllers, Services and Repositories!
- Rather, you'll often see collaboration between multiple instances:
  - One Controller triggering operations in several Services
  - One Service accessing several Repositories
  - One-to-many relationships between Entities
  - Entities that are used in several Services
  - Entities without a dedicated controller
  - …
- Common distribution:
  - Lots of controllers (one for every kind of feature / request the users can make)
  - Many services (depending on how closely different features are related)
  - A few repositories (one for each entity)





To Autowire or Not to Autowire?

- Dependency injection via autowiring, and the need for adding an interface to enable this, may appear to make the implementation unnecessarily complex. However:

- **Autowiring is necessary** when involving a JPA repository
  - The JPA repository implementation is auto-generated by the JPA, so you need autowiring to obtain its instance in your service
  - This means you can’t instantiate your service yourself, but need autowiring to obtain its instance in your controller
- **Autowiring is recommended** even for simple services not using a repository
  - You could create an instance of the service yourself in the controller whenever you need it
  - But in a high-volume application, the creation and destruction of all those objects is costly
  - Better to use autowiring, so the Java Spring Framework is in charge of service instantiation and can handle this in an efficient manner (e.g. through using an instance pool)

Assignment 3: Content

- By **Sun 28 Oct**, submit as PDF document(s) in Ugla:
  - **A UML class diagram** defining the structure of your project:
    - Showing the classes your final system shall consist of
      - with attributes, data types, methods, relations, multiplicities
  - A **UML sequence diagram** illustrating the control flow:
    - Showing the collaboration of JSPs, controllers, services, repository, and potential client-side components in one exemplary request-response cycle
      - Make sure the method calls in the sequence diagram reflect the methods specified in the class diagram
- On **Thu 1 Nov**, explain the diagrams of your design model to your tutor:
  - Why did you choose particular solutions?
  - Which aspects of the system’s behavior are particularly complex?
  - Where do you see room for design improvement (if you had time after the semester)?
  - How are the MVC pattern and the design principles of encapsulation, separation of concerns etc. reflected in your design (especially if you have client-side code)?

Assignment 3: Grading Criteria

- **UML Class Diagram** (50%)
  - ✓ clean, syntactically correct UML!
  - ✓ includes your controllers, application logic, persistence logic, data structures
  - ✓ clearly shows attributes, methods, relationships, multiplicities of classes
  - ✓ reflects MVC pattern, design principles of encapsulation, separation of concerns etc.

- **UML Sequence Diagram** (50%)
  - ✓ clean, syntactically correct UML!
  - ✓ shows how your controller, application logic and data model classes collaborate (i.e. call each other) in a request-response cycle
  - ✓ clearly shows relevant call structures (methods and parameters)

Design Models

see also:

- Larman: Applying UML and Patterns, Ch. 17 & 25

Domain Models vs. Design Models

Focus of next lecture

- We create **domain models** in order to **understand the application domain**
  - So we can talk to the business stakeholders about their requirements on their own terms
  - So we can introduce new team members to the purpose of the software before introducing them to its implementation

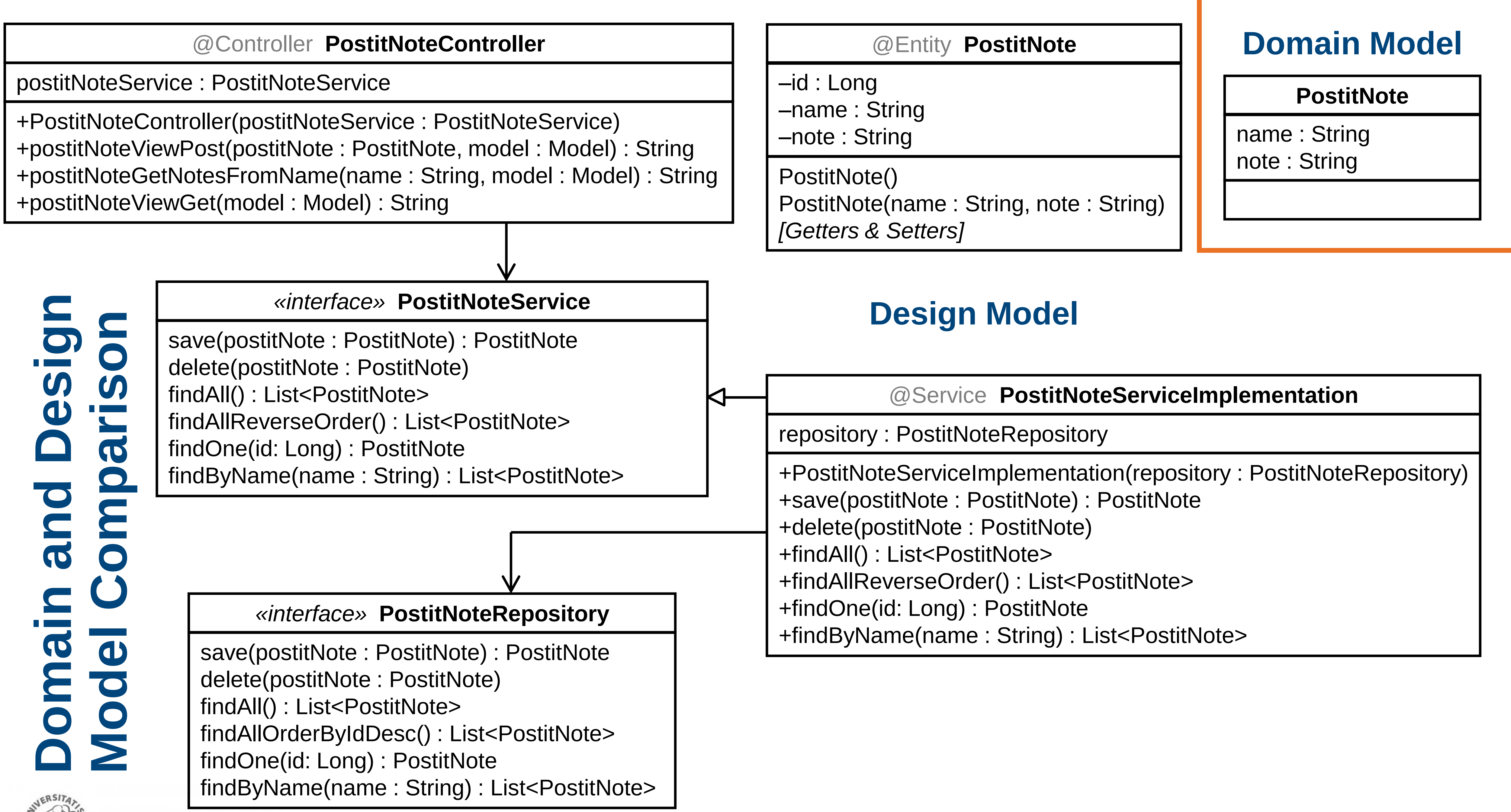
➢ Domain models illustrate business aspects / requirements / **problem domain**

- We create **design models** in order to express **how our solution shall work**
  - So we can talk to fellow developers
    - about what the best technical solutions to a given domain problem would be
    - about the concrete structures that we plan to implement
  - So we can introduce new team members to the internal structure/mechanics of the software after they have understood the application domain

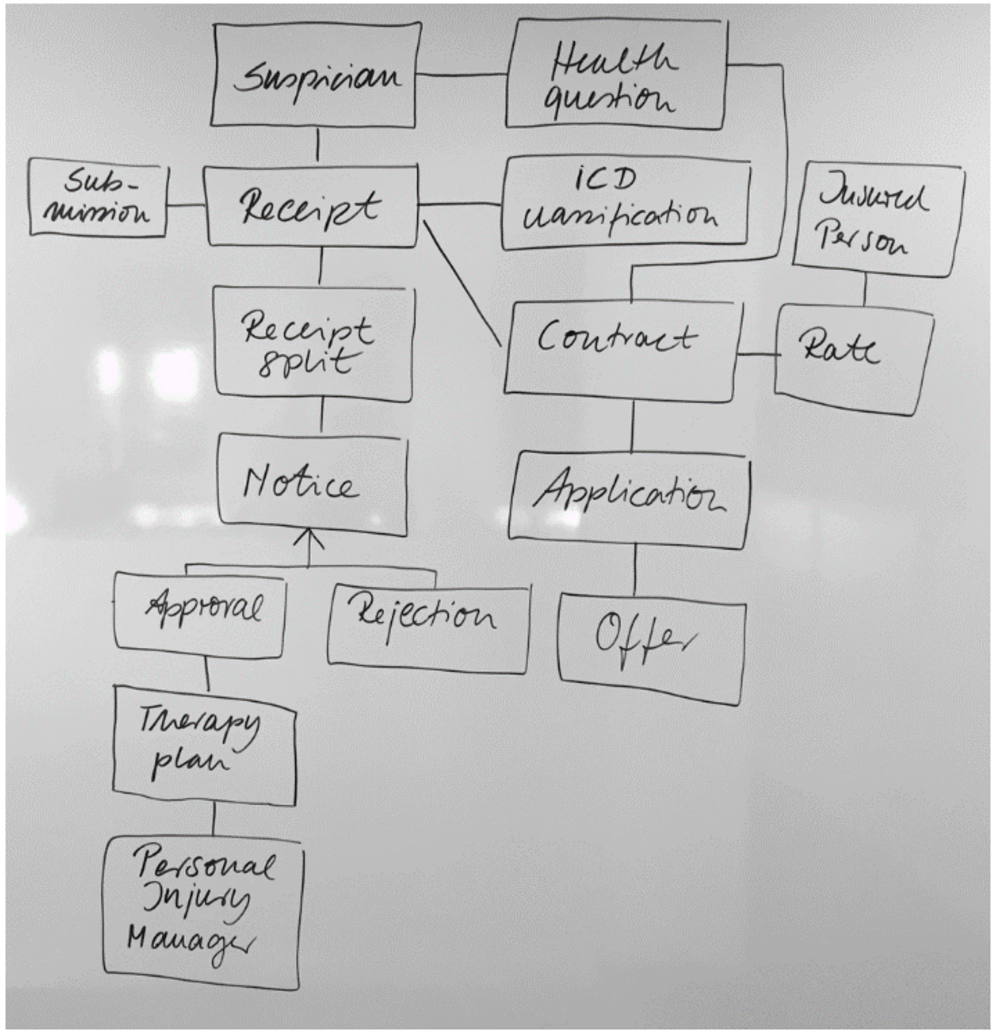
➢ Design models illustrate technical aspects / implementation / **solution domain**

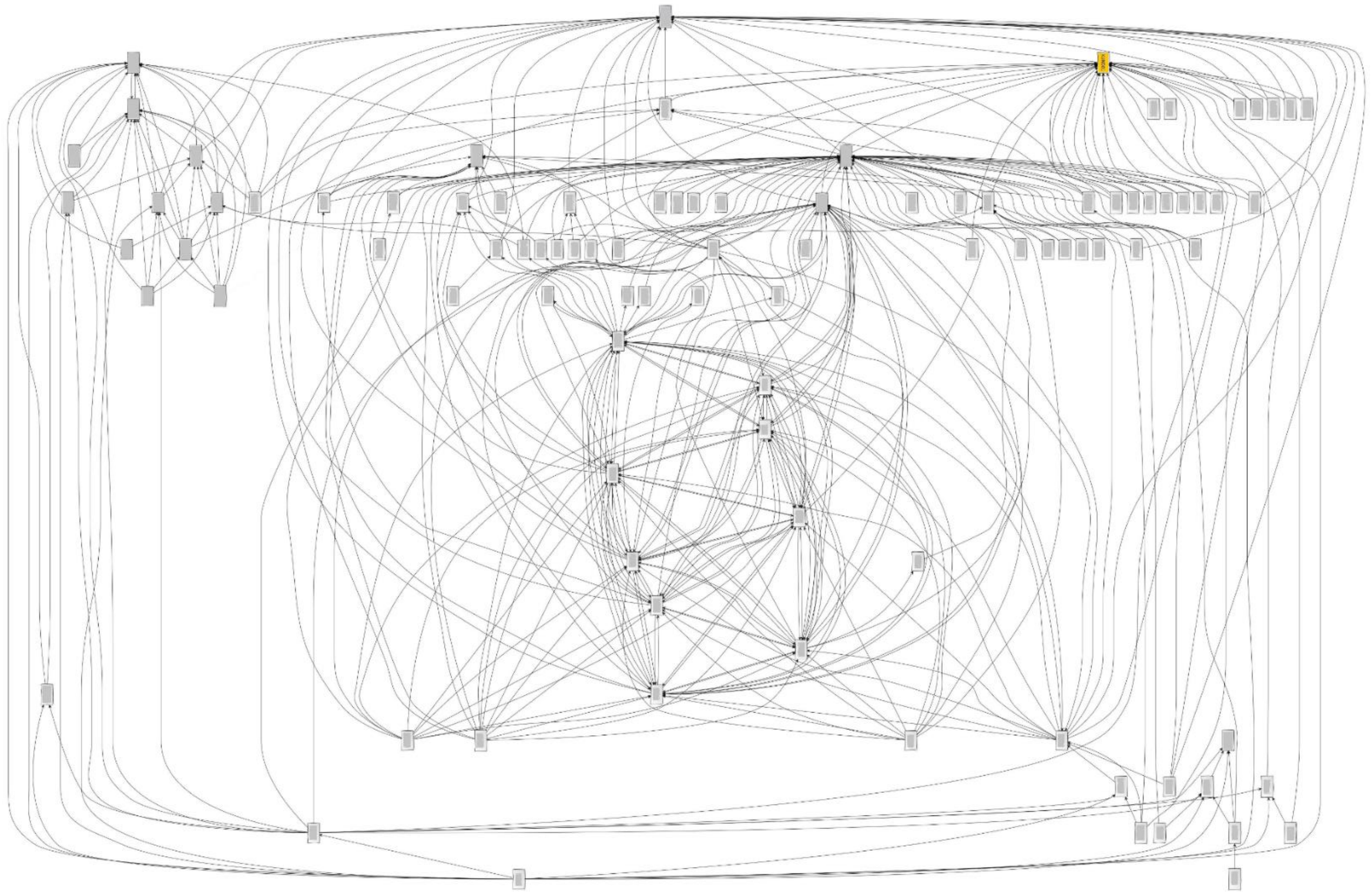
Focus of this lecture and Assignment 3

Domain and Design Model Comparison


Domain Model
PostitNote
name : String
note : String
Design Model
@Controller PostitNoteController
postitNoteService : PostitNoteService
+PostitNoteController(postitNoteService : PostitNoteService)
+postitNoteViewPost(postitNote : PostitNote, model : Model) : String
+postitNoteGetNotesFromName(name : String, model : Model) : String
+postitNoteViewGet(model : Model) : String
@Entity PostitNote
–id : Long
–name : String
–note : String
PostitNote()
PostitNote(name : String, note : String)
[Getters & Setters]
«interface» PostitNoteService
save(postitNote : PostitNote) : PostitNote
delete(postitNote : PostitNote)
findAll() : List<PostitNote>
findAllReverseOrder() : List<PostitNote>
findOne(id: Long) : PostitNote
findByName(name : String) : List<PostitNote>
@Service PostitNoteServiceImplementation
repository : PostitNoteRepository
+PostitNoteServiceImplementation(repository : PostitNoteRepository)
+save(postitNote : PostitNote) : PostitNote
+delete(postitNote : PostitNote)
+findAll() : List<PostitNote>
+findAllReverseOrder() : List<PostitNote>
+findOne(id: Long) : PostitNote
+findByName(name : String) : List<PostitNote>
«interface» PostitNoteRepository
save(postitNote : PostitNote) : PostitNote
delete(postitNote : PostitNote)
findAll() : List<PostitNote>
findAllOrderByIdDesc() : List<PostitNote>
findOne(id: Long) : PostitNote
findByName(name : String) : List<PostitNote>

More Complex Domain Model Examples





120 classes, 6 million customers with associated data objects

From Domain Models to Design Models

- Design models are not merely an incremental refinement of domain models.

- Actors in domain model may be mere data objects in design model
  - e.g. a Buyer driving a real-life sales process
- Passive objects in domain model may have responsibilities in design model
  - e.g. a Sale object calculating the total of its LineItems
- Some elements of domain model may not be part of design model
  - e.g. the User is still part of the whole system, but “in the flesh” rather than as an object
- Some necessary parts of the design model may not be in the domain model
  - e.g. objects describing types rather than instances; data structures; technical components…

- The design model is inspired by the domain model, but not derived from it.

Object-Oriented Design Heuristics

Guidelines / considerations for constructing the design model:

A. Identifying the most suitable creator of objects of a particular class *(→ L17.10)*
B. Identifying the most suitable objects to fulfill certain responsibilities *(→ L17.11)*
C. Expressing support concepts not present in the application domain *(→ L25.2)*
D. Keeping objects' responsibilities focused *(→ L17.14)*
E. Keeping objects independent from other objects *(→ L17.12)*
F. Protecting other objects from variation/instability within an object *(→ L25.4)*
G. Expressing variations in object behavior *(→ L25.1)*
H. Decoupling objects even if they need to work closely together *(→ L25.3)*
I. Assigning responsibility for reacting to user input *(→ L17.13)*

A. Identifying Most Suitable Creators of Objects
(Creator Principle)

- **Problem**
  - In the application domain, many objects might simply exist *a priori*
  - But they need to be explicitly created in the software
  - Sometimes, we may even need to create various representations of them
    - e.g. a general `ProductDescription`, and an actual `Product` that can be put in a user's `Sale`

- **Recommendation**
  - Objects of class A should create objects of class B if
    - A "contains" or compositely aggregates B
    - A records or closely uses B
    - A has initializing data that needs to be passed to B upon its instantiation
      - i.e. A is an "Expert" for B

A. Identifying Most Suitable Creators of Objects
(Creator Principle)

- **Contraindications**
  - If creation requires particular complexity, e.g. …
    - recycling instances for performance reasons
    - conditionally creating instances from a family of related classes
  - …creation should be delegated to a Factory helper class *(→ HBV401G)*

- **Note**
  - Creators are often (but not always) identical with Experts.

B. Identifying Suitable Objects to Fulfill Responsibilities
(Expert Principle)

- **Problem**
  - Domain and design objects are best understood as having certain responsibilities.
  - Which objects should implement which responsibilities?
  - May be hard to decide when several objects collaborate to fulfill a responsibility in the application domain

- **Recommendation**
  1. Clearly state what is the responsibility to assign to some class
  2. Identify the class that has most of the information pertinent to that responsibility
     1. Look for such a class in the design model first
     2. If there is no such class, derive an appropriate class from the domain model
  3. Assign the responsibility to that class

B. Identifying Suitable Objects to Fulfill Responsibilities
(Expert Principle)

- **Contraindications**
  - Expertise shouldn’t be the only criterion in assigning responsibility to classes; coupling and cohesion need to be considered as well
    - e.g. a data object shouldn’t be responsible for displaying itself in the user interface or storing itself in the database

- **Note**
  - Even objects that are “passive” in the application domain can serve active roles as Experts and Creators of classes in the technical design and implementation.

C. Expressing Support Concepts Not Found in Domain
(Fabrication Principle)

- **Problem**
  - Sometimes, there is no suitable domain-model class to which a certain responsibility can be assigned in the design model
  - e.g. responsibilities for organizing data into structures or for performing technical / support operations such as persistence
- **Recommendation**
  - Add classes to the design model as needed, even if they do not have a counterpart in the domain model, especially
    - technical classes such as `PersistentStorage`, `Controller` etc.
    - structural classes such as `List`, `Map`, `Tree` etc.
    - representational classes such as a `ProductDescription` for a `Product`
    - behavioral classes such as `TableOfContentsGenerator`
- **Contraindications**
  - Do not overdo behavioral decomposition – not every algorithm needs its own class

D. Keeping Objects' Responsibilities Focused
(High Cohesion Principle)

- **Problem**
  - How to ensure that objects remain focused, understandable, reusable, maintainable…
  - …and that they are stable and not constantly exposed to change?
- **Recommendation**
  - When assigning responsibilities to classes (based on the other principles), ensure that the responsibilities of a class are strongly related to each other.
- **Contraindications**
  - Under some circumstances, it may be necessary to give up cohesion with regard to one functional aspect in order to gain it for another functional aspect
    - e.g. bundling all persistent storage access in one class (high cohesion wrt. storage), even if that class is used to store many different kinds of data (low cohesion wrt. data types)

E. Keeping Objects Independent From Other Objects
(Low Coupling Principle)

- **Problem**
  - How to ensure that objects are as independent as possible from other objects, that they are largely unaffected by outside changes, and can be reused?
- **Recommendations**
  - When assigning responsibilities to classes (based on other principles), ensure that the class is connected to, has knowledge of, or depends on other elements as *little* as possible.
  - ➢ Create the following connections only if necessary:
    - Class A has an attribute referring to objects of class B
    - Class A has a method referring to objects of class B in parameters, local variables or return type
    - Class A calls a method of class B
    - Class A is a direct or indirect subclass of class B
    - Class A implements the interface B

E. Keeping Objects Independent From Other Objects
(Low Coupling Principle)

- **Recommendations (cont'd.)**
  - "Don't talk to strangers" principle:
    Within an object's method, try to access only attributes and methods of:
    - `this` object itself
    - A parameter of the method
    - An attribute of `this` object
    - An element of a collection that is an attribute of `this` object
    - An object created within the method
  - Avoid (within reason) long chains of reference resolutions
    - e.g. `obj.getX().getY().getZ().method()`

- **Note:** Relax – it's impossible to follow all these recommendations strictly
  - Objects do need to talk to each other
  - Just make sure their communication is focused and reasonable

E. Keeping Objects Independent From Other Objects
(Low Coupling Principle)

- **Contraindications**
  - Some degree of coupling is natural since objects are *supposed* to collaborate
  - Keep coupling at a reasonable level
    - If coupling is too high, classes will be exposed to changes elsewhere
    - If coupling is too low, classes will be stuffed with responsibilities, violating High Cohesion principle
  - Coupling to stable and pervasive elements (e.g. in the Java class libraries) is fine
    - e.g. long chains of method calls are common when working with streams

- **Notes**
  - High coupling is not a problem per se – the problematic part is coupling to elements that are *unstable* (in their interface, behavior, presence…)
    - ➢ Avoid coupling to unstable elements
  - But don’t “future-proof” your design excessively or spend effort on minimizing coupling when there is no realistic stability threat

F. Protecting Other Objects From Variation/Instability
(Protected Variations Principle)

- **Problem**
  - How to ensure that variations or instabilities within a class that we are designing do not have undesirable impact on other classes?

- **Recommendations**
  - Identify points of predicted variation or instability
  - Create a stable interface around them that hides any internal changes

- **Contraindications**
  - It makes sense to protect external objects against *known* variation points in our object
  - But consider if it’s worth the effort to prophylactically protect against *possible* evolution points

- **Note**
  - This is a fundamental principle of software design (“information hiding” – D. Parnas, 1972)

G. Expressing Variations in Object Behavior
(Polymorphism Principle)

- **Problem**
  - How to implement alternative behaviors depending on the “type” of a domain object?
  - How to exchange service implementations without having to inform the calling layer?
- **Recommendations**
  - Encapsulate alternative behaviors in different subclasses of the same superclass or interface
  - Refer to the implementation only through the superclass/interface
  - Do not make the type of a domain object explicit in an attribute, and don’t test for it and use conditional logic to perform different behaviors
    - “Type” is a major built-in concept of object-oriented languages – use it, don’t simulate/rebuild it!
  - Prefer using `interfaces`, esp. when you don’t need to inherit existing implementations, or when you want to combine independent characteristics. Use abstract classes otherwise.
- **Notes**
  - A number of common design patterns rely on polymorphism, e.g. Adapter, Command, Composite, Proxy, State, Strategy

H. Decoupling Objects That Would Be Closely Coupled
(Indirection Principle)

- **Problem**
  - How to avoid direct coupling between several classes that rely closely on each other, but should not or cannot communicate directly with each other?
- **Recommendation**
  - Assign responsibility for facilitating the communication to an intermediate object that abstracts away from the characteristics the objects should not expose to each other.
    - e.g. the JPA acts as an intermediary between business logic and persistence logic, allowing both layers to work closely together without having to be aware of each other's implementation details
- **Notes**
  - A number of common design patterns rely on indirection, e.g. Adapter, Bridge, Façade, Observer, Mediator

I. Assigning Responsibility for Reacting to User Input
(Controller Principle)

- **Problem**
  - The user interface layer should only be concerned with displaying the user interface (UI).
  - The business layer should only be concerned with executing business logic, without being aware of the UI.
  - Who determines what happens upon certain UI events, and what to display upon certain business logic results?
- **Recommendations**
  - Have a dedicated control layer separating UI and business logic
  - Have several controllers – use either of these approaches to structure them:
    - A **façade controller** handles everything related to a particular subsystem or component
    - A **use case controller** handles all aspects of a particular use case
  - Let controllers receive and **prepare data** as necessary for processing by either layer
  - Let controllers **delegate control** to business layer for actual processing
  - Make sure you don't end up with one bloated (low cohesion) controller doing everything!

In-Class Quiz #6: Object-Oriented Design Heuristics

Which goals (a-h) do the principles (1-8) work toward?

a) Assigning responsibility for reacting to user input
b) Decoupling objects even if they need to work closely together
c) Expressing support concepts not present in the application domain
d) Expressing variations in object behavior
e) Identifying the most suitable creator of objects of a particular class
f) Keeping objects independent from other objects
g) Keeping objects' responsibilities focused
h) Protecting other objects from variation/instability within an object

1. Creator
2. Controller
3. Fabrication
4. High cohesion
5. Indirection
6. Low coupling
7. Protected variations
8. Polymorphism