CptS 322- Software Engineering Principles I

Software Design and Architecture

Instructor: Sakire Arslan Ay

Fall 2021

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

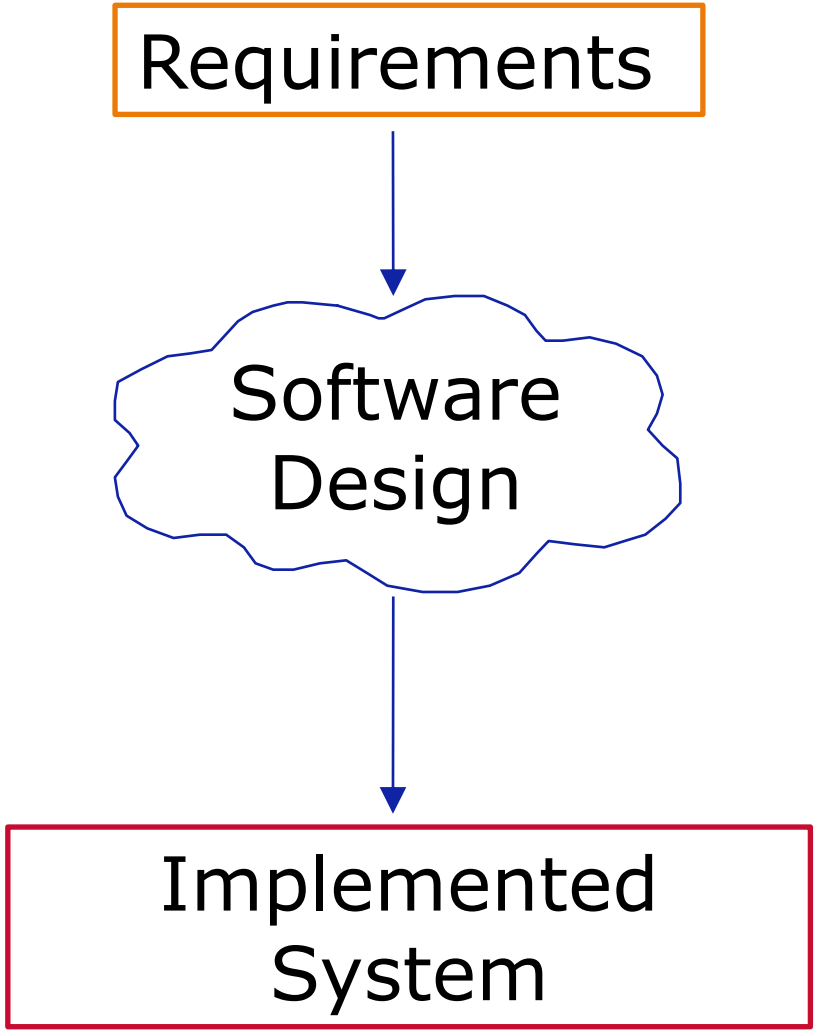
Software Design

Definition:

- *Design* is a problem-solving process whose objective is to find and describe a way:
  - to implement the system's *functional requirements*...
  - while respecting the constraints imposed by the *quality, platform, and process requirements*...
    - including the budget
  - and while adhering to general principles of *good quality*

The Basic Problem: From Requirements to Code

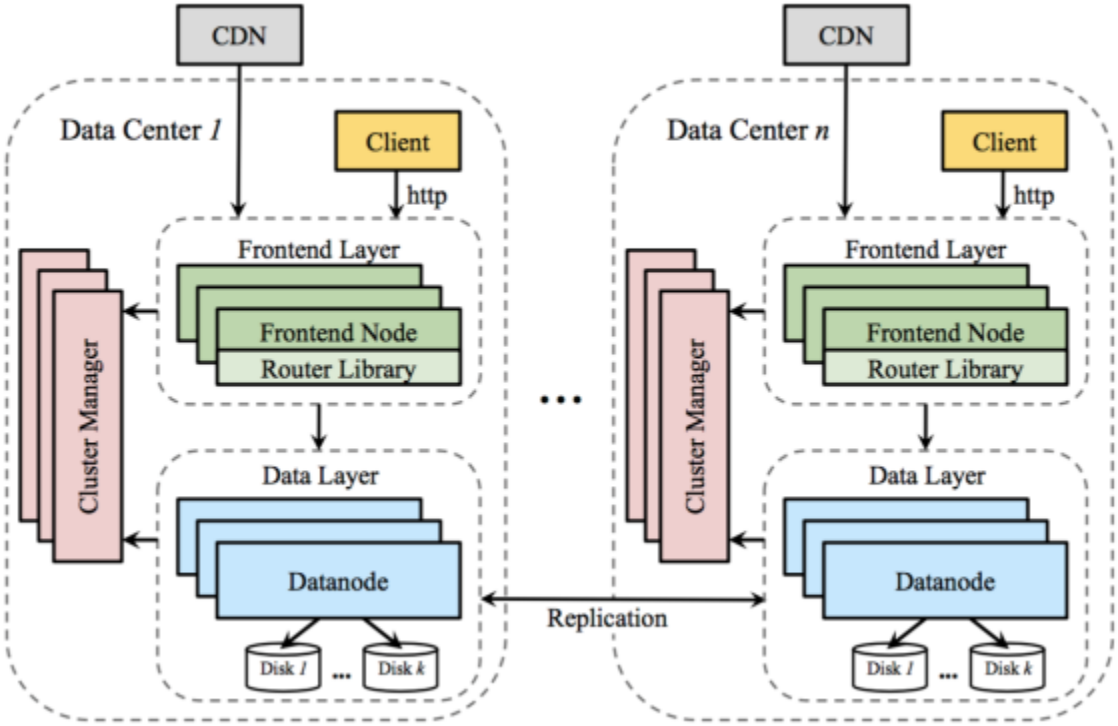
- Bridge the gap
  - between a problem and an implemented system in a manageable way



Different Aspects of Design

- *Architecture design*:
  - The division into subsystems and components,
    - How these will be connected.
    - How they will interact.
    - Their interfaces.
- *Class design*:
  - The various features of classes.
- *User interface design*
- *Algorithm design*:
  - The design of computational mechanisms.
- *Protocol design*:
  - The design of communications protocol.

Software Architecture



Ambry: LinkedIn's scalable geo-distributed object store Noghabi et al. SIGMOD '16

What does a box represent? An arrow? A layer? Adjacent boxes?

Models for Software Architecture

- Our models for software design and architecture will be based on UML
- UML = Unified Modeling Language
- A standard notation for describing (drawing) software design
  - Also implementation details such as subclassing, interfaces (dependencies), and much more
- Widely used in industry
- We will cover:
  - UML component diagram (subsytems)
  - UML deployment diagram
  - UML class diagram (classes)
  - UML sequence diagram (behavior)
  - UML state diagram (behavior)

Tools for UML

- Visual:
  - draw.io (https://www.draw.io)
  - Star UML or White Star UML
- PlantUML
  - Free, small, enough features
  - Describe diagrams in textual source, compile to image
  - And diagrams are generated automatically from source

Software Architecture: subsystems

- Trying to deal with something big all at once is normally much harder than dealing with a series of smaller things

- So, divide the system into subsystems
  - Each individual component is smaller, and therefore easier to understand.
  - Parts can be replaced or changed without having to replace or extensively change other parts.
  - Separate people that can work on each part.

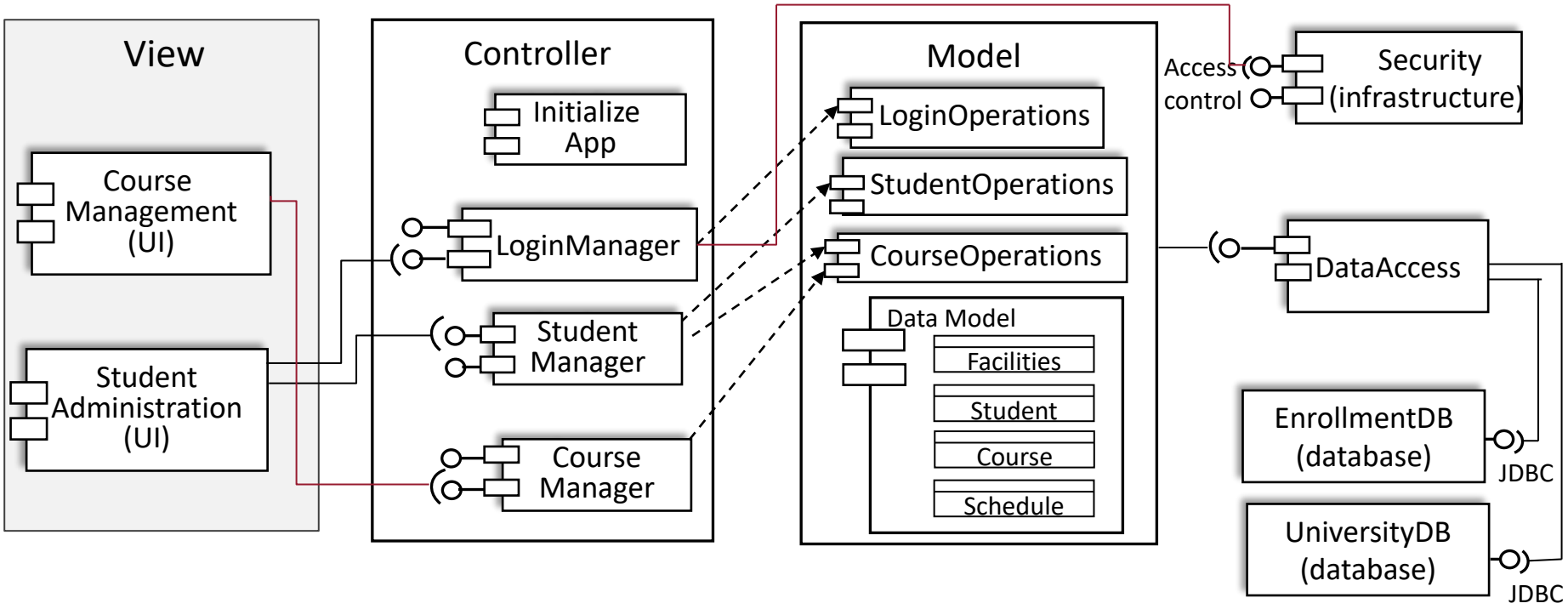
Subsystems and Classes

- Subsystem (UML: Component)
  - A replaceable part of the system with well-defined interfaces
  - Collection of classes, associations, operations, events that are closely interrelated with each other
    - Seed for subsystems: UML Objects and Classes.
  - Several programming languages provide constructs for modeling subsystems,
    - packages in java,
    - namespaces in C#
  - In other languages (for example C, C++) subsystems are not explicitly modeled

Ways of dividing a software system into subsystems

- A distributed system is usually divided up into clients and servers
- A system is divided up into subsystems that encapsulates the state and the behavior of related classes
- If a subsytem is too complex, it can be further decomposed into simpler subsystems.

UML Component Diagram



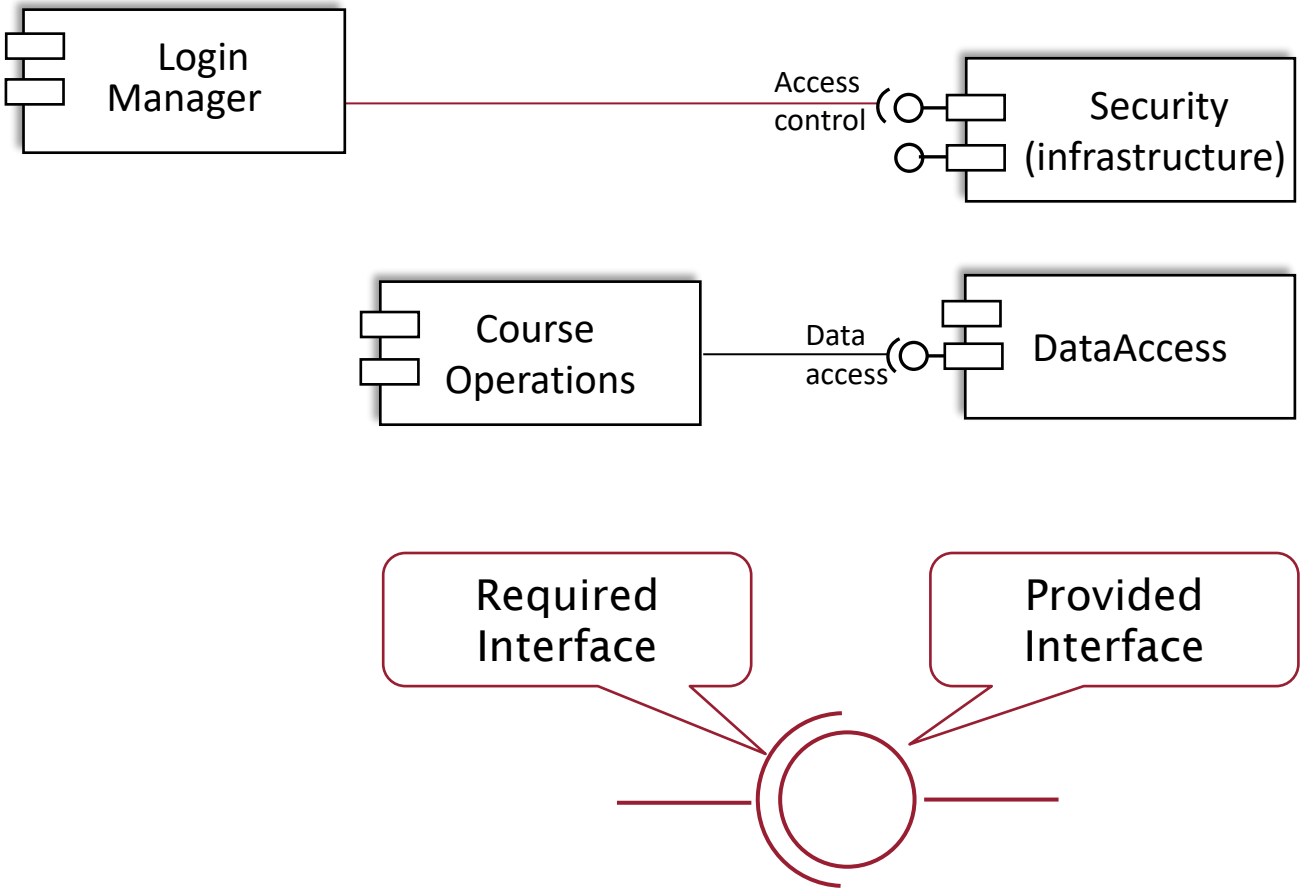
- Subsystem decomposition for university course management system
- Subsystems are shown as UML components
- Dashed arrow indicate “use” (dependencies)
- Socket-lollipop connections represent the provided/requested services

Services and Subsystem Interfaces

- Service
  - Named operations
  - The origin (“seed”) for services are the use cases from the requirements
  - A subsystem is characterized by the services it provides to other subsystems

- Subsystem interface:
  - The set of operations and services of a subsystem available to other subsystems
  - Specifies the interaction and information flow from and to subsystem boundaries, but not inside the subsystem
  - Should be well-defined and small
  - Should minimize the information on implementation.

Services and Subsystem Interfaces



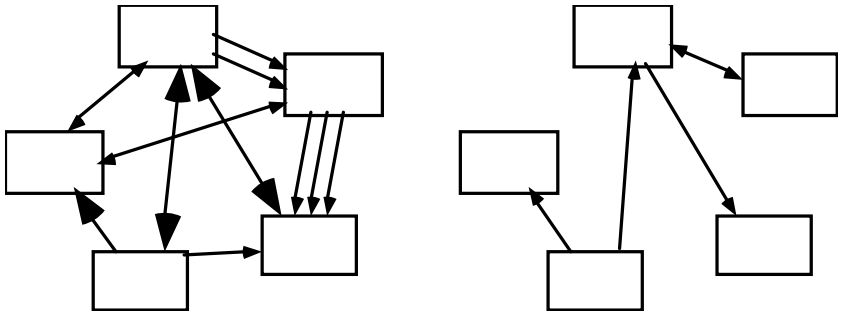
- Ball-socket notation showing provided and required interfaces

Design principles of software architecture:

1. Reduce coupling where possible
2. Increase cohesion where possible
3. Keep the level of abstraction as high as possible
4. Increase reusability where possible
5. Reuse existing designs and code where possible
6. Design for flexibility
7. Anticipate changes
8. Design for portability
9. Design for testability

Design Principle-1 : Reduce coupling where possible

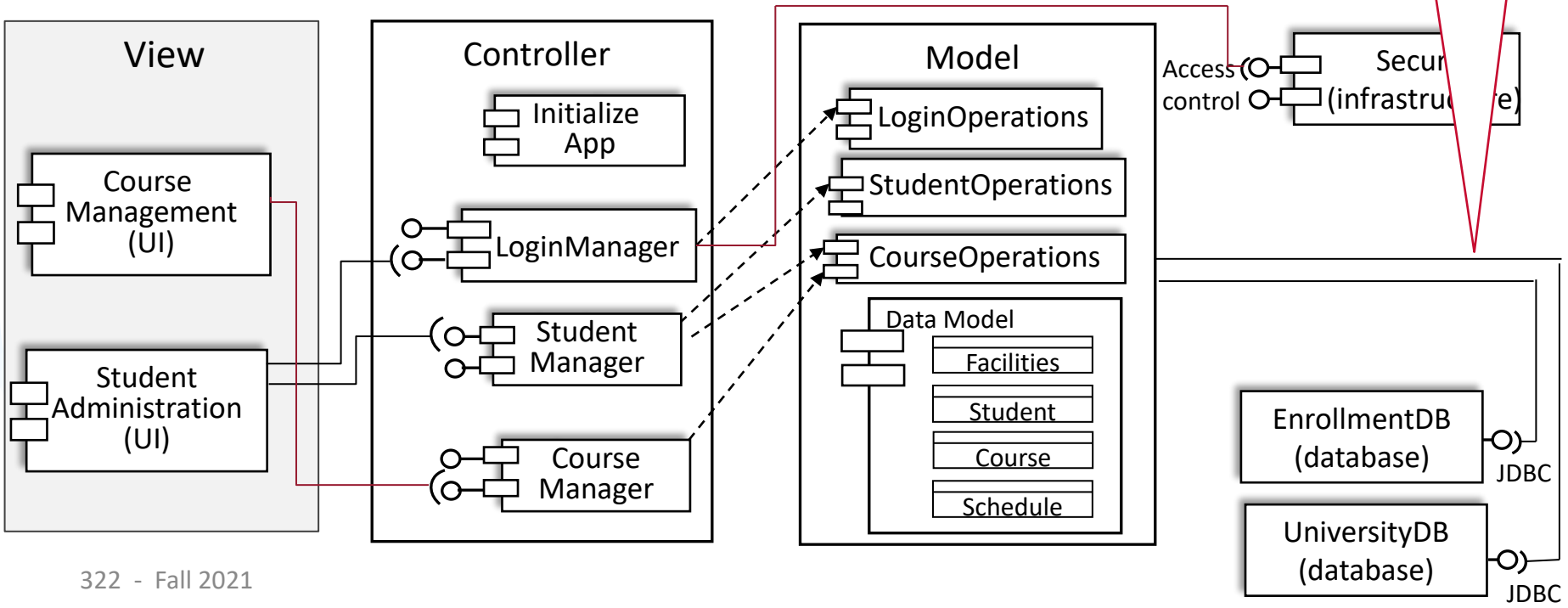
- *Coupling* measures *interdependencies* between one subsystem and another
  - High coupling: Modifications to one subsystem will have high impact on the other subsystem (change of model, massive recompilation, etc.).
  - Low coupling: The two systems are relatively independent from each other.



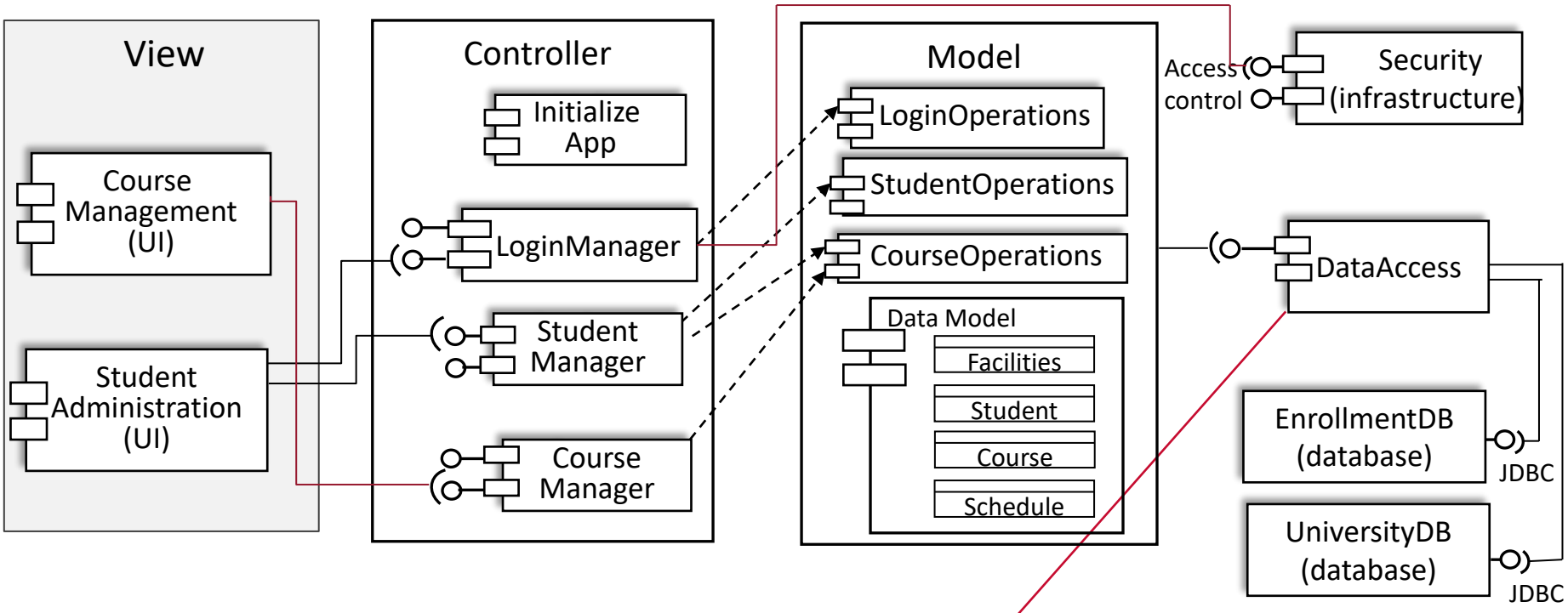
How to achieve low coupling

- Low coupling can be achieved if a calling class does not need to know anything about the internals of the called class
  - Principle of information hiding

Does this architecture design achieve lower coupling compared to the original design (see next slide)?



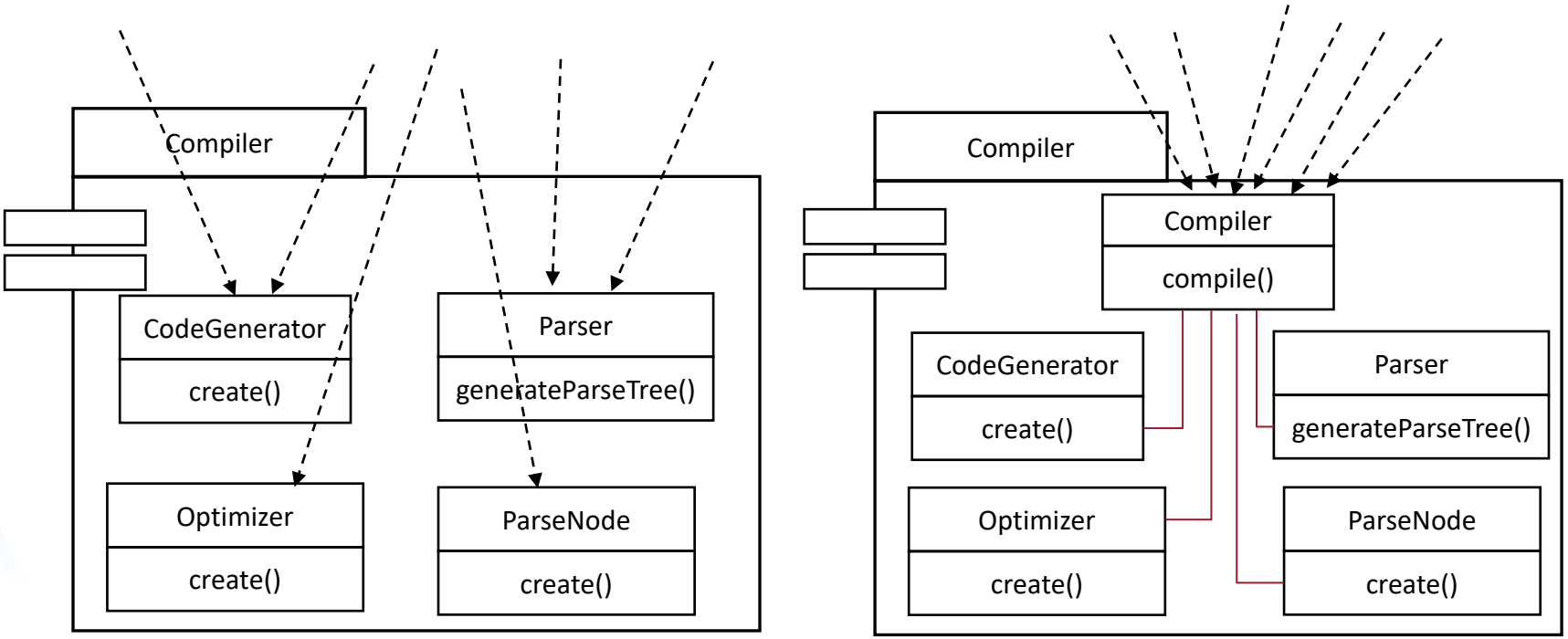
How to achieve low coupling- Example



We reduce coupling by introducing the *DataAccess* subsystem/component.
LoginOperations, StudentOperations, and CourseOperations subsystems will not be affected by the changes in the databases.

Reducing coupling: Use design patterns

- The Façade design pattern allows us to further reduce dependencies between classes
  - Encapsulates a subsystem with a simple, unified interface.



Design Principle-2: Increase cohesion where possible

- A subsystem or module has high cohesion if it keeps together things that are related to each other, and keeps out other things.
  - This makes the system as a whole easier to understand and change
  - High coherence: The classes in the subsystem perform similar tasks and are related to each other (via associations)
  - Low coherence: Lots of miscellaneous and auxiliary objects, no associations

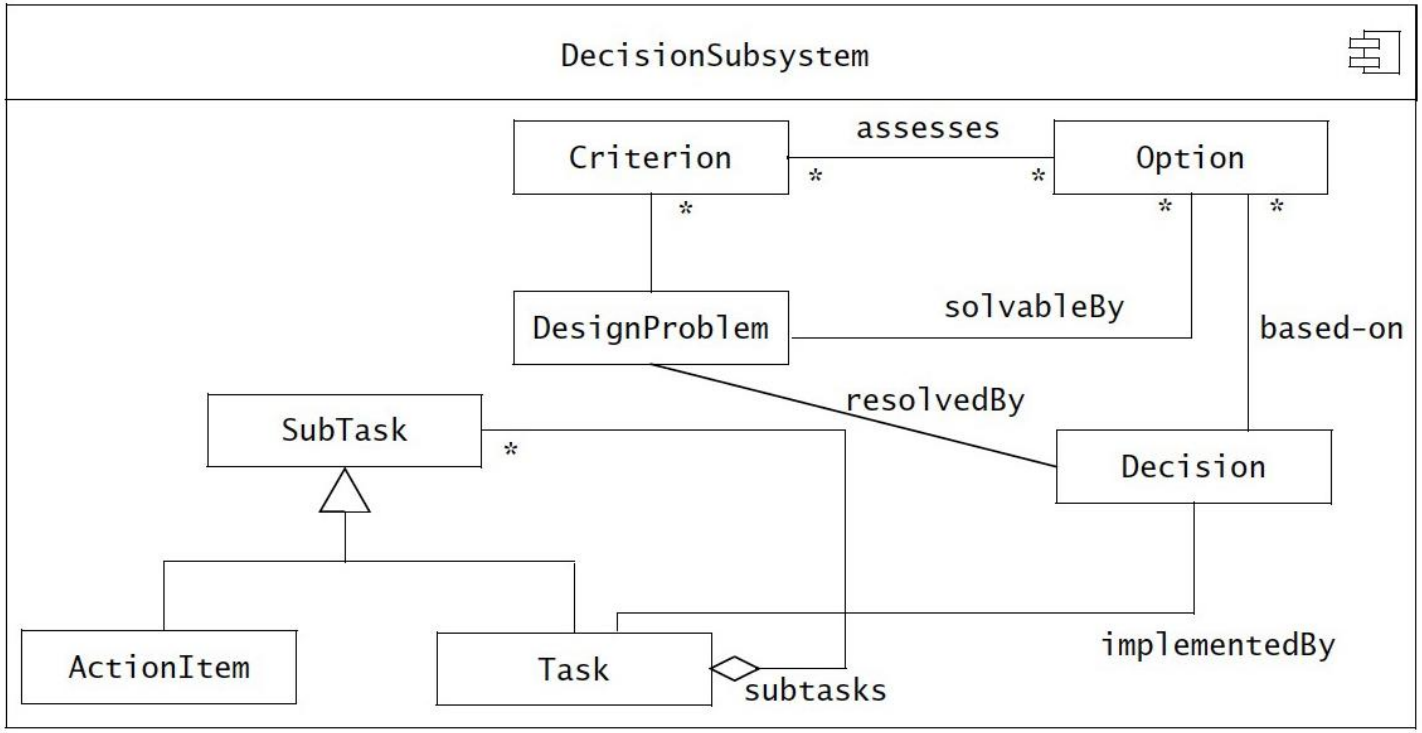
How to achieve high cohesion

- High coherence can be achieved if most of the interaction is within subsystems, rather than across subsystem boundaries

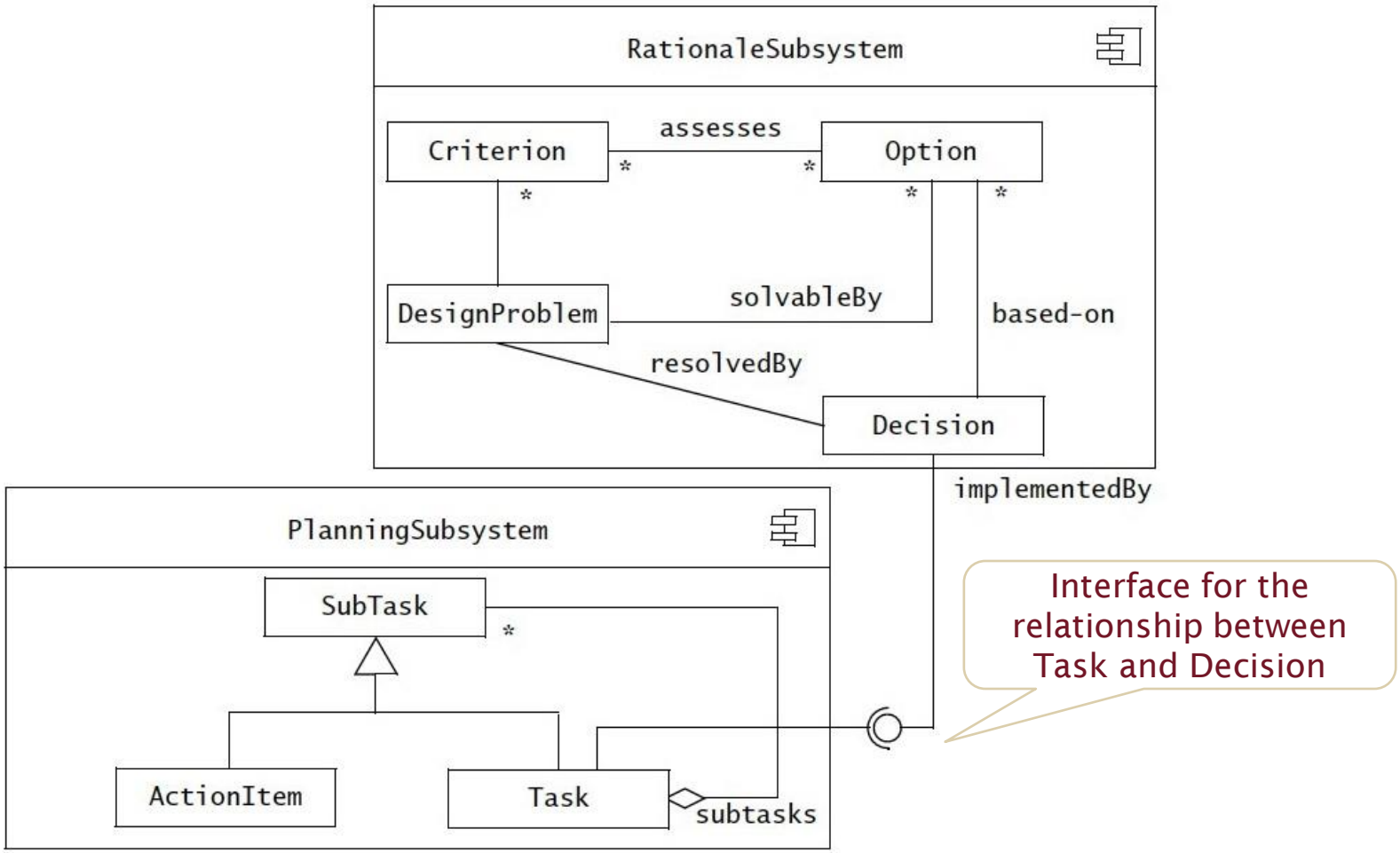
- Questions to ask:
  - Does one subsystem always call another one?
    - Yes: Consider moving them together into the same subystem.
  - Which of the classes in a subsystem call each other?
    - Can the cohesion be improved by restructuring the subsystems or changing the subsystem interface?
  - Can the subsystems even be hierarchically ordered (in layers)?

Example of increasing cohesion in a subsystem

- Can the cohesion of “DecisionSubsystem” be improved?



Example of increasing cohesion in a subsystem



- Alternative subsystem decomposition. The cohesion new subsystems is higher than the original subsystem.

Design Principle 3: Keep the level of abstraction as high as possible

- Ensure that your designs allow you to hide or defer consideration of details, thus reducing complexity
  - A good abstraction is said to provide information hiding
  - Abstractions allow you to understand the essence of a subsystem without having to know unnecessary details

Design Principle 4: Increase reusability where possible

- Design the various aspects of your system so that they can be used again in other contexts
  - Generalize your design as much as possible
  - Follow the preceding three design principles
  - Simplify your design as much as possible

Design Principle 5: Reuse existing designs and code where possible

- Design with reuse is complementary to design for reusability
  - Actively reusing designs or code allows you to take advantage of the investment you or others have made in reusable components
    - *Cloning* should not be seen as a form of reuse

Design Principle 6: Design for flexibility

- Actively anticipate changes that a design may have to undergo in the future, and prepare for them
  - Reduce coupling and increase cohesion
  - Create abstractions
  - Do not hard-code anything
  - Leave all options open
    - Do not restrict the options of people who have to modify the system later
  - Use reusable code and make code reusable

Design Principle 7: Anticipate changes

- Plan for changes in the technology or environment so the software will continue to run or can be easily changed
  - Avoid using early releases of technology
  - Avoid using software libraries that are specific to particular environments
  - Avoid using undocumented features or little-used features of software libraries
  - Avoid using software or special hardware from companies that are less likely to provide long-term support
  - Use standard languages and technologies that are supported by multiple vendors

Design Principle 8: Design for portability

- Have the software run on as many platforms as possible
  - Avoid the use of facilities that are specific to one particular environment
  - E.g., a library only available in Microsoft Windows

Design Principle 9: Design for testability

- Take steps to make testing easier
  - Discussed more in CptS 422
  - Ensure that all the functionality of the code can be tested independently
  - Write automated tests

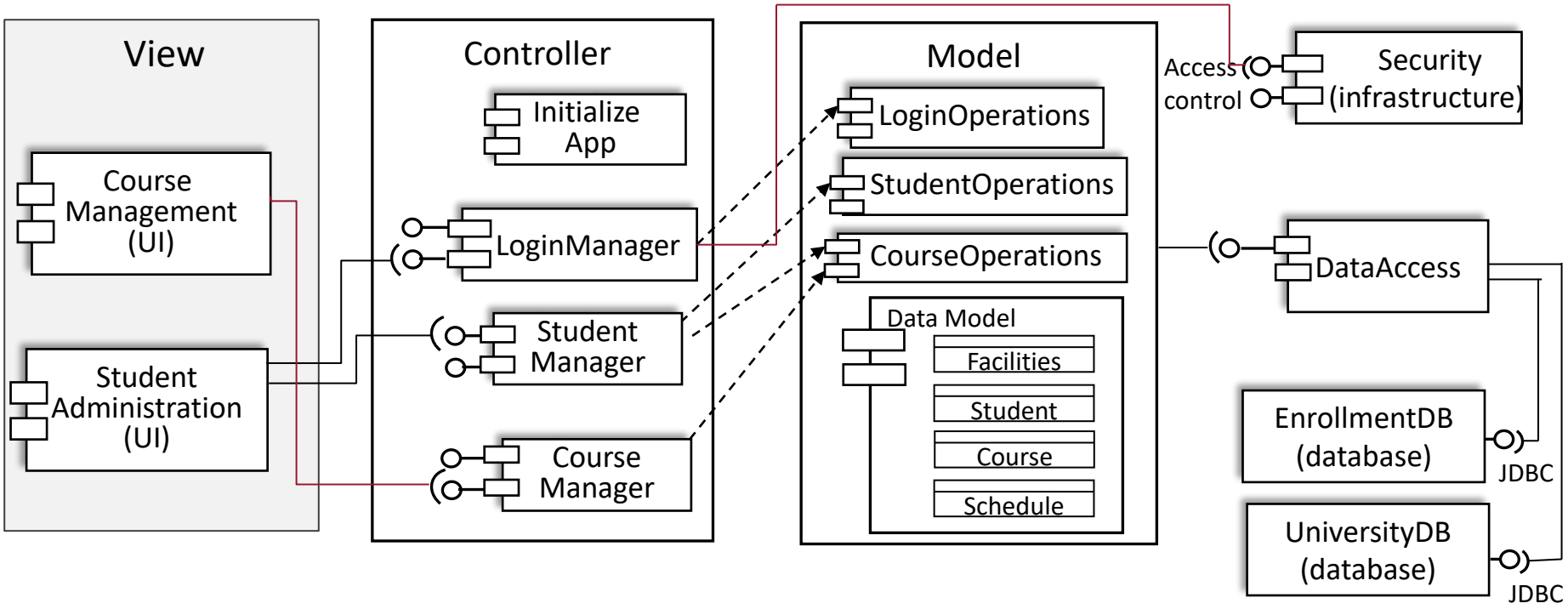
Developing an architectural model

1. Start by sketching an outline of the architecture
   - Based on the principal requirements and use cases, determine the main components that will be needed
   - Choose among the various architectural patterns
     - Discussed next
   - *Suggestion*: have several different teams independently develop a first draft of the architecture and merge together the best ideas

Developing an architectural model

2. Refine the architecture
   - Identify the main ways in which the components will interact and the interfaces between them
   - Decide how each piece of data and functionality will be distributed among the various components
   - Determine if you can re-use an existing framework, if you can build a framework
3. Consider each use case and adjust the architecture to make it realizable
4. Mature the architecture

UML Component Diagram - revisited



- Subsystem decomposition for university course management system
- Subsystems are shown as UML components
- Dashed arrow indicate “use” (dependencies)
- Socket-lollipop connections represent the provided/requested services

Deployment Diagrams

- Physical allocation of components to computational units.
- Nodes: computational units
- Edges: communication between these units

Deployment Diagrams