Object-Oriented Design II

SWEN-261
Introduction to Software Engineering

Department of Software Engineering
Rochester Institute of Technology

Controller

Pure fabrication

Open/close

Polymorphism

Liskov substitution

The lesson continues building your object-oriented design skills with several other design principles.

- These are also principles from SOLID and GRASP

SOLID

- **S**ingle Responsibility
- **O**pen/closed
- **L**iskov substitution
- **I**nterface segregation
- **D**ependency inversion

GRASP

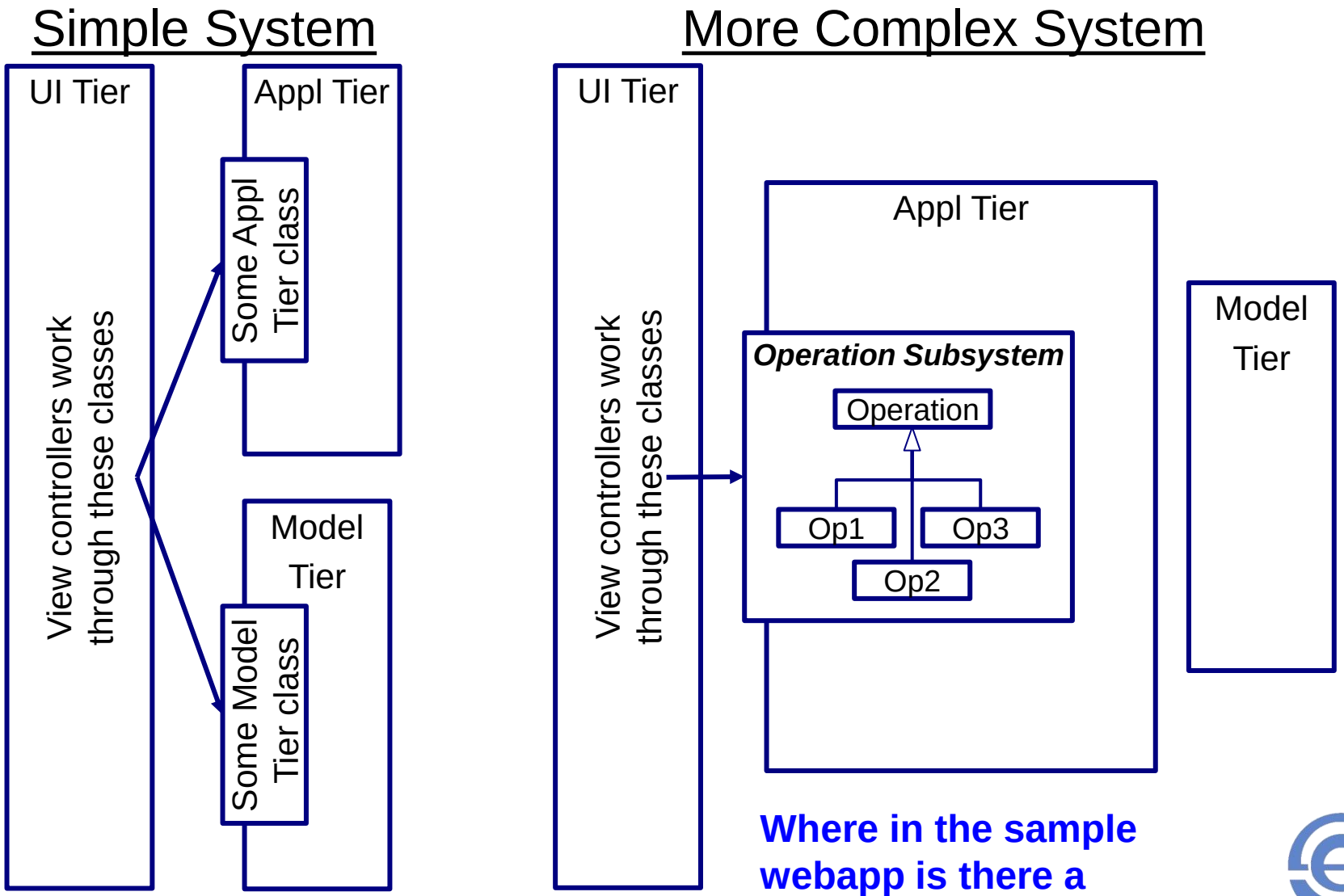
- Controller
- Creator
- Indirection
- Information expert
- High cohesion
- Low coupling
- Polymorphism
- Protected variations
- Pure fabrication

Controller specifies a separation of concerns between the UI tier and other system tiers.

> *Assign responsibility to receive and coordinate a system operation to a class outside of the UI tier.*

- "Controller" is an overused term in software design.
  - ***In GRASP, this is <u>not</u> the view "controller" which is firmly in the UI tier.***
- In simple systems, it may be a single object that coordinates all system operations.
- In more complex systems, it is often multiple objects from different classes each of which handles a small set of closely related operations.

Here is how GRASP controllers fit into the software architecture.



Where in the sample webapp is there a GRASP-style controller?

Pure Fabrication is sometimes needed to balance other design principles.

Assign a cohesive set of responsibilities to a non-domain entity in order to support high cohesion and low coupling.

- Your design should be primarily driven by the problem domain.
- To maintain a cohesive design you may need to create classes that are not domain entities.
- In the previous slide, the Operation Subsystem was a pure fabrication.

What were pure fabrications in the sample webapp? How could you have implemented it without those fabrications?

The *Open/closed* principle deals with extending and protecting functionality.

Software entities should be open for extension, but closed for modification.

- Software functionality should be extendable without modifying the base functionality.
  - ***Mostly provided by features of implementation language: inheritance, interface***
- Your design should consider appropriate use of
  - ***Inheritance from abstract classes***
  - ***Implementation of interfaces***
- Dependency injection provides a mechanism for extending functionality without modification.

Polymorphism creates a hierarchy when related behavior varies by class.

Assign responsibility for related behavior that varies by class by using polymorphic behavior.

- Polymorphism is a primary object-oriented concept and should be used whenever possible
- Bad code smells that indicate a potential class hierarchy and use of polymorphism
  - ***Conditional that selects behavior based on a "type" attribute***
  - ***Use of `instanceof` or similar language constructs to select operations to perform***

The *Liskov substitution* principle constrains the pre- and post-conditions of operations.

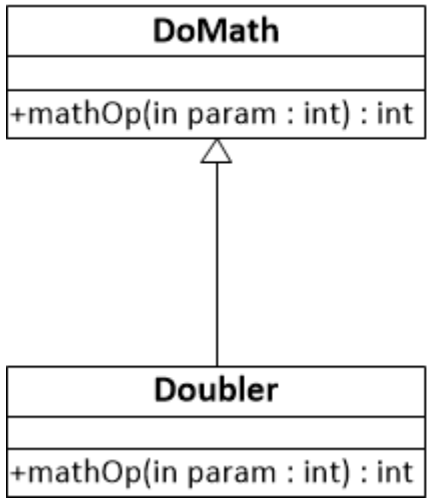
> *Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program*

- Pre-conditions specify what must be true before a method call.
- Post-conditions specify what will be true after a method call.
- Design by Contract is a programming technique that requires formal definition of the pre- and post-conditions and has language support for it.

Any subclass of a class should be able to substitute for the superclass without error.

- A subclass must not violate any of the pre- or post-conditions guaranteed by the superclass.
- Superclass clients count on the pre- and post-conditions being true even when polymorphism has the client interacting with a subclass.
- To maintain a pre-condition, a subclass must not narrow the pre-condition, i.e. be a subset.
- To maintain a post-condition, a subclass must not broaden the post-condition, i.e. be a superset.

Here is what Liskov substitution allows.



mathOp()

Precondition: param can be between 1 and 10. mathOp() will not fail with input in that range.

Postcondition: return value is guaranteed to be between 0 and 30. The client will fail if it is outside of that range.

If client reference is to:

DoMath

Doubler

mathOp() must accept a param of 1 through 10.

It could accept a wider range, i.e. 1 to 15.

mathOp() can not quadruple the value of the parameter because that would lead to a broader post-condition, i.e. return value between 4 and 40.

mathOp() could have a narrower post-condition of 3 to 17, i.e. it is a PlusTwo class for its full range of input.