Summary of the course lectures

Components and Interfaces

- Components:
  - Compile-time: Packages, Classes, Methods, …
  - Run-time: Objects, Invocations,
- Interfaces:
  - What the client needs to know:
    - Syntactic and Semantic
    - About instances of this particular class
    - About all instances of this class and its descendants (LSP)
- Levels of Description
  - Architectural, Package, Class, Method

UML

- Class Diagrams
  - Classes and Interfaces
  - Association
    - Plain Aggregation Composition
    - Navagbility
    - Multiplicity
  - Specialization and Realization
  - Dependence

UML

- Sequence Diagrams
  - The tree of calls.
  - Just an example of behaviour

Use Cases

- Like a movie script
  1. Ilsa: … But of course that was the day the Germans marched into Paris.
  2. Rick: Not an easy day to forget.
  3. Ilsa: No
  4. Rick: I remember every detail. The Germans wore gray, you wore blue.

  5. Customer: Inserts her bank card belonging to our bank
  6. ATM: Swallows card and prompts for PIN
  7. Customer : Types in PIN and presses OK
  8. ATM: Shows the main menu.

Use Cases

- Describe “Happy day Scenarios”, but also alternatives.
  - 3a: The customer enters a PIN and presses cancel
  - 4a: The card is returned and the use case ends.

Use Cases

- Can be used
  - to verify external design with customer
  - as a basis for testing
  - to sequence iterations
  - to drive development

Classes and Responsibilities

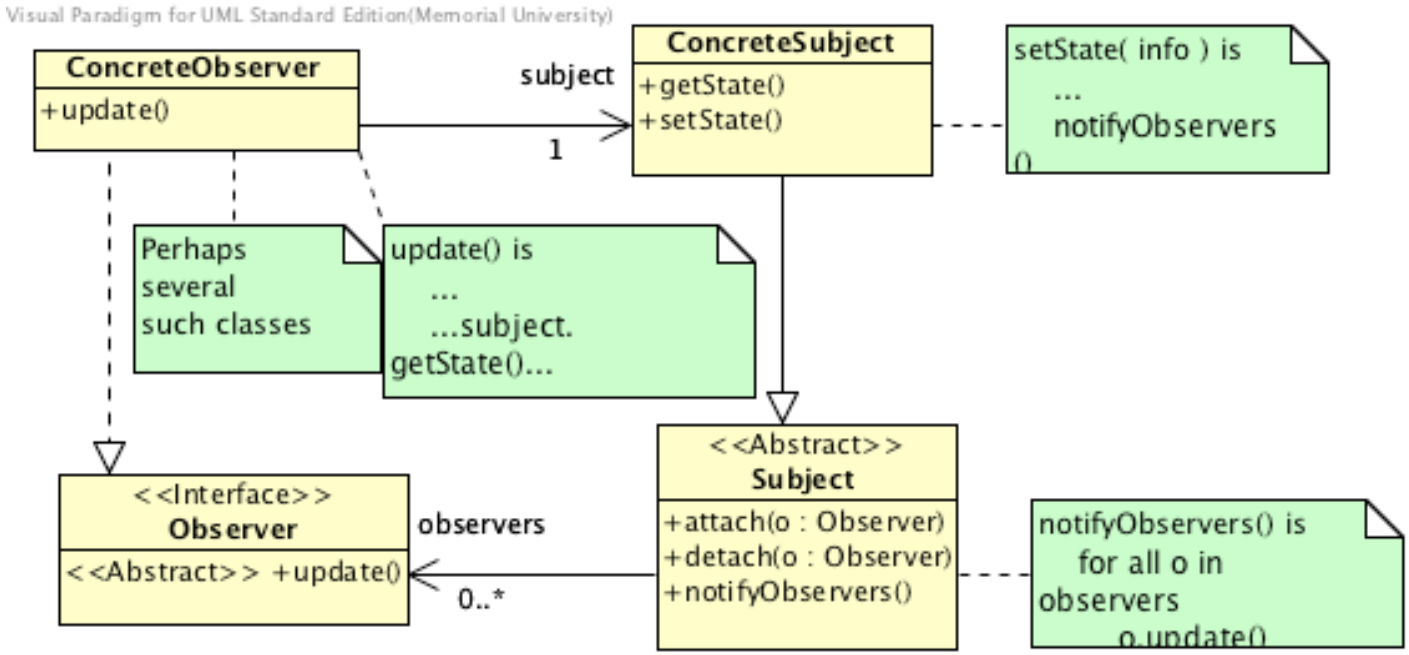
- Each class has responsibilities that are
  - Specific, Few, & Easily understood
- Kinds of responsibilities
  - Information Storage, Algorithmic, Device Interface, OS Interface, I/O formatting, Adaptation, Structural, Creators, Arbitrary Facts.
- Information Hiding
  - Classes can hide design decisions and external constraints from the rest of the system.
  - Ideally each axis of change is confined to one class (Information Hiding Principle)
  - And each class deals with one axis of change (SRP)

Model / View

- Analysing the problem of Models and Views we found a way to protect the model from
  - changes to the number of views
  - changes to the class(es) of the views
  - specific requirements of specific views
- And to protect the views from
  - implementation details of the model such as data storage methods
  - the need to inform other views of changes to the model

Observer Pattern

- Our solution to the Model / View problem turns out to have a name and be more widely applicable.
- It is the observer pattern:

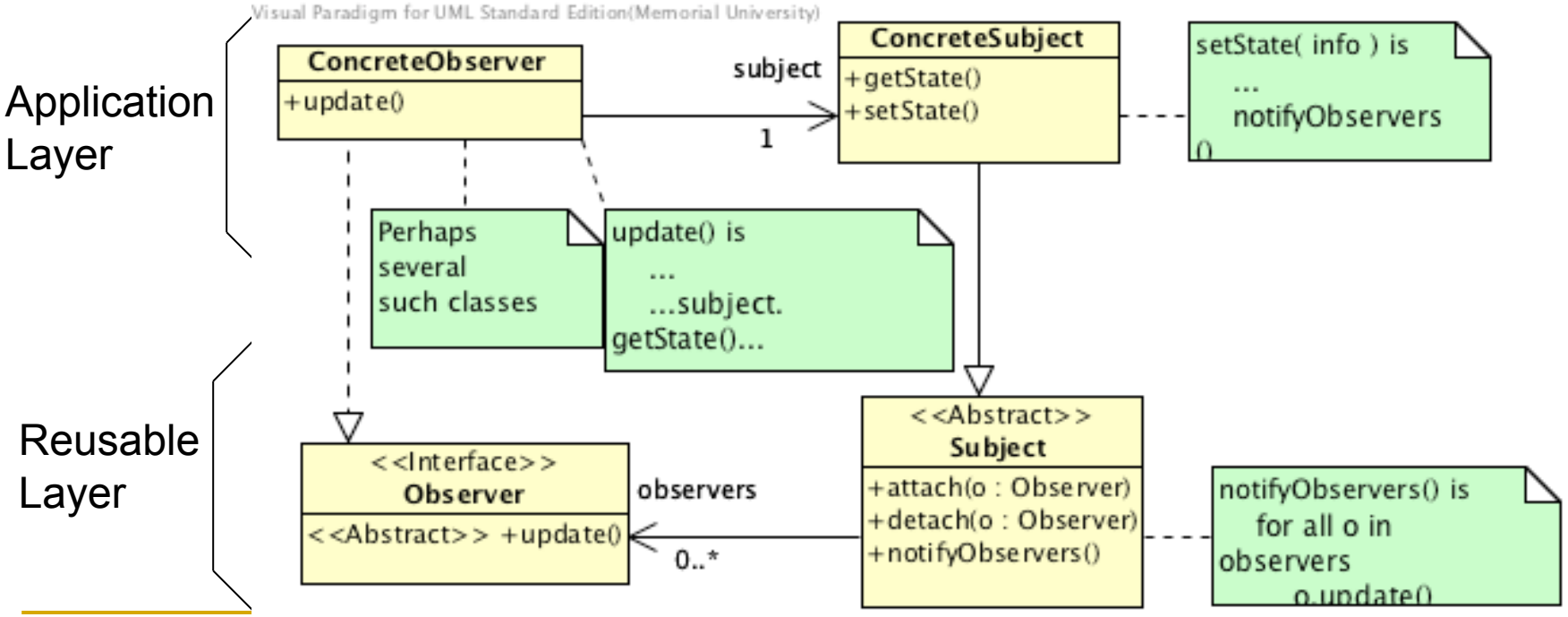


Design Patterns

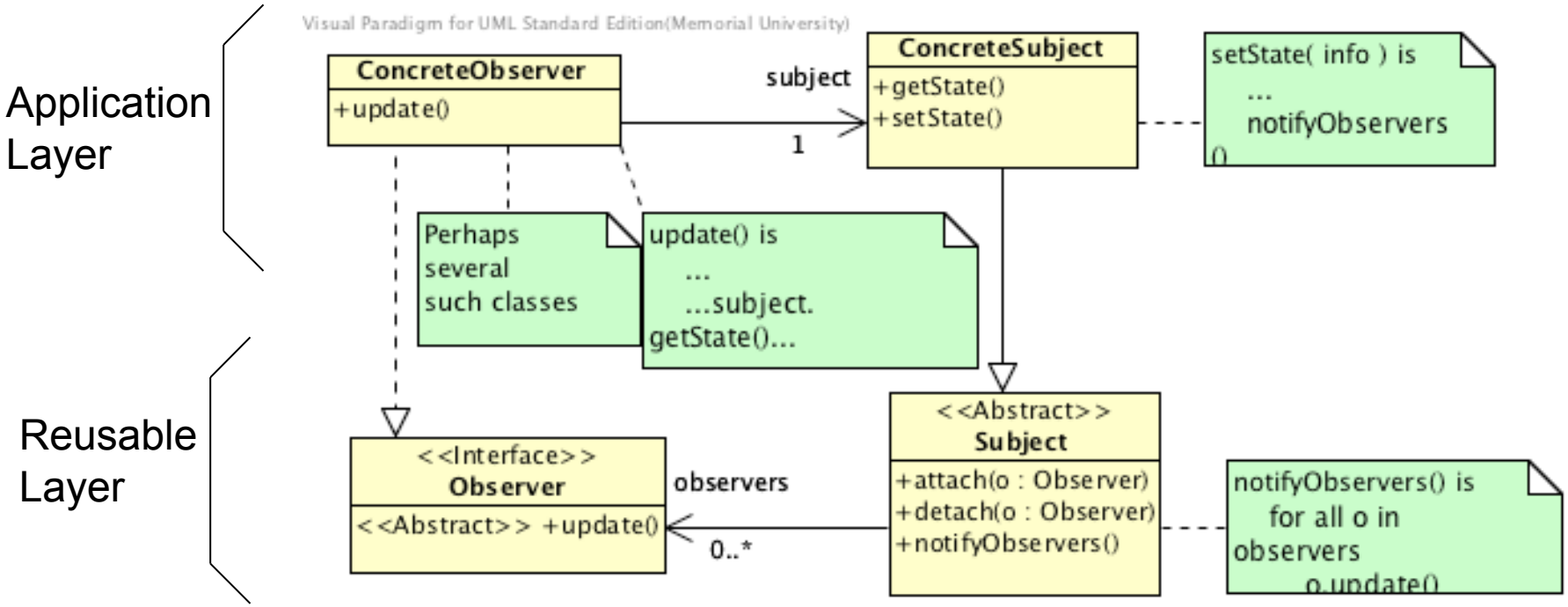
- Design patterns are reusable solutions to recurring problems in software design
- Many patterns are identified and discussed in Gamma, Helm, Johnson, & Vlissides, *Design Patterns: Elements of Reusable OO Software*, AW, 1994.
- Standard Format: Intent, Also known as, Motivation, Applicability, Structure, Collaborations, Consequences, Implementations, and Variations, Known Uses

Design Patterns

- Recurring theme: Design patterns structure classes to minimize dependence and thus allow aspects to be reusable. E.g.



Design Patterns



- Even if you don't plan to reuse an aspect of the design, this "separation of concerns" simplifies your software. (SRP, ISP)

Observer Pattern

- Separates the concern of observer notification from the implementation of the model and the implementation of the views.

Command Pattern

- Separates “what needs to be done” from “when to do it”.

Composite Pattern

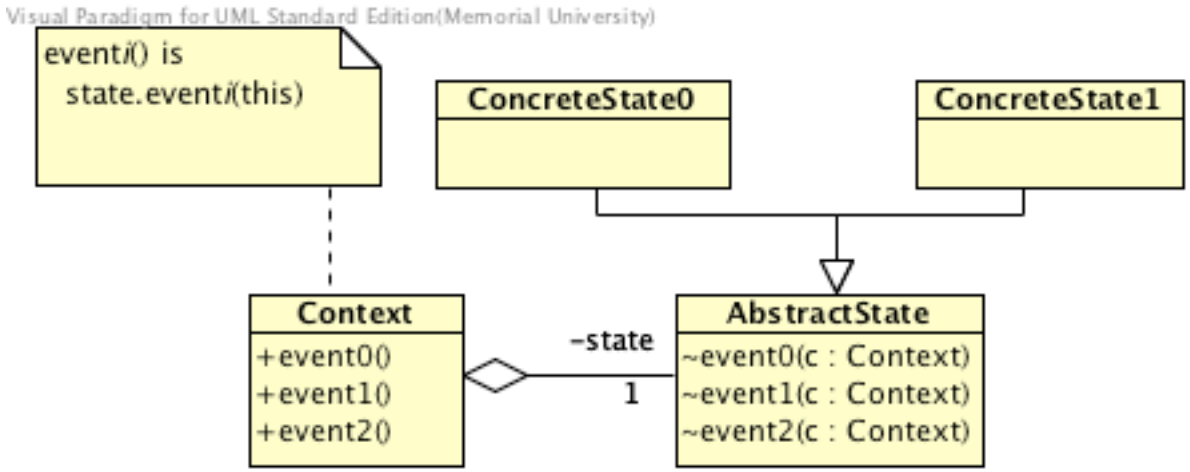
- Compose objects into trees structures to represent part-whole hierarchies
  - Calls are sent down the tree

Proxy Pattern

- Provides a surrogate or placeholder for another object
  - The Subject object and the Proxy object implement the same interface.
  - The Proxy may hide the true location or other aspects of the Subject
  - [Full disclosure: My notes actually describe both the Decorator pattern and the Proxy pattern, but use only the name "Proxy pattern". Here's the difference: When new functionality is added by the intermediate object, it's the Decorator pattern. When location or other aspects of the subject are hidden by the intermediate object, it's the Proxy pattern. The JSnoopy example is properly a case of the Decorator pattern. Adding to the confusion, Java's `Proxy` class is often used to automate implementation of the Decorator pattern. See Gamma *et al.* or Freeman and Freeman for more.]

State Pattern

- Represents each state in a state machine by a class.
  - Events are mapped to method calls.
  - The context keeps a pointer to the current state and delegates events to its current state



Factory Method Pattern

- Create objects via an interface
  - Different implementations will produce different products.
  - Relieves the client of having to depend on the concrete class of the product in order to create it

    ```
    AbstractProduct p = new ConcreteProduct() ;
    ```

    is replaced by

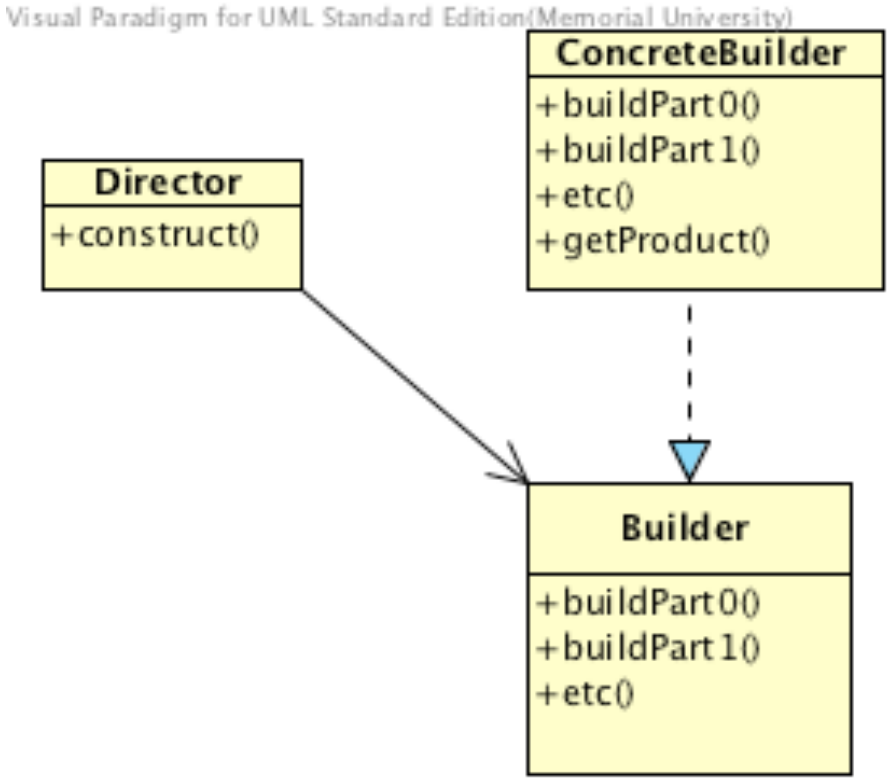
    ```
    AbstractProduct p = creator.create() ;
    ```
  - In effect the creator is a *strategy* for creating products.

Builder Pattern

- Reuse a parser (or other client) with multiple representations.
  - The interface between the parser and the representation is defined.
  - A generalization of the Factory Method. Whereas, in the factory method, it only takes one call to build the product, with the builder, multiple calls are used in sequence to build the product.

The Builder Pattern

- The Builder object typically may be the product.



Other Patterns

- Abstract Factory
  - Create families of objects via an interface
- Singleton
  - Ensure a class only has one instance
- Adapter
  - Change the interface without changing funtionality
- Façade
  - Provide a single interface to a set of objects

Other Patterns

- Iterator
  - Give sequential access to the items of a collection without exposing its representation
- Mediator
  - Use an object to mediate all interaction between two or more other objects
- Template Method
  - Vary details of algorithm by filling in abstract methods
- Strategy
  - Configure objects with strategy objects that encapsulate different policies

Single Responsibility Principle (SRP)

- "A class should only have one reason to change"
  - Consequently most changes will affect only a small proportion of your classes

Open / Closed Principle

- "Software entities (classes, modules, functions etc) should be open for extension but closed for modification"
  - Design software entities so that future changes on an axis require no modification
  - Consider the template method pattern and strategy pattern

Refactoring

- Respond to change by first improving the structure of the software so that the portion that is not affected is isolated from change on the axis. Then accommodate the change.
- I.e. first restructure according to the SRP, the IHP, and the OCP.

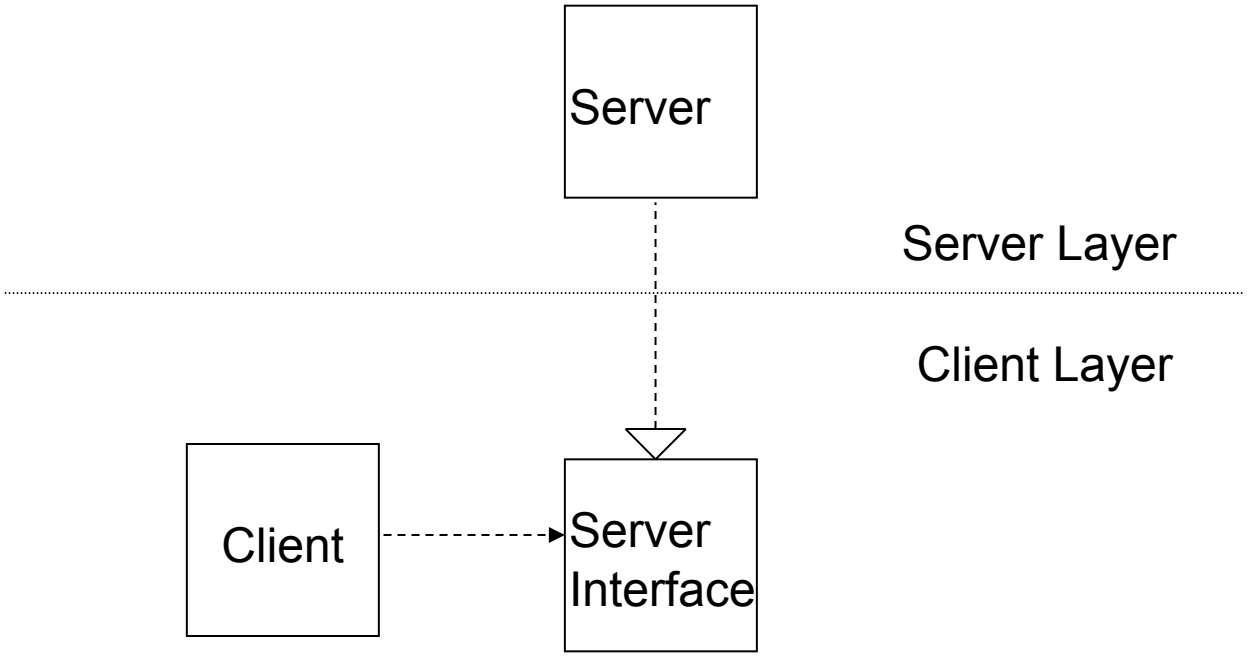
Liskov Substitution Principle (LSP)

- If S is a subtype of T, objects of type S should behave as objects of type T, if they are treated as objects of type T
  - Each type T has two interfaces
    - Direct Interface: What the client can expect of x, if x is known to be a direct instance of T
    - Indirect Interface: What the client can expect of x, if x is known to be a (possibly indirect) instance of T
  - The LSP can be understood as saying that the code of S should respect the indirect interface of S and all supertypes T of S
  - The direct interface should be documented, but may (usually) be discovered from the code,
  - The indirect interface must be documented.

Dependence Inversion Principle (DIP)

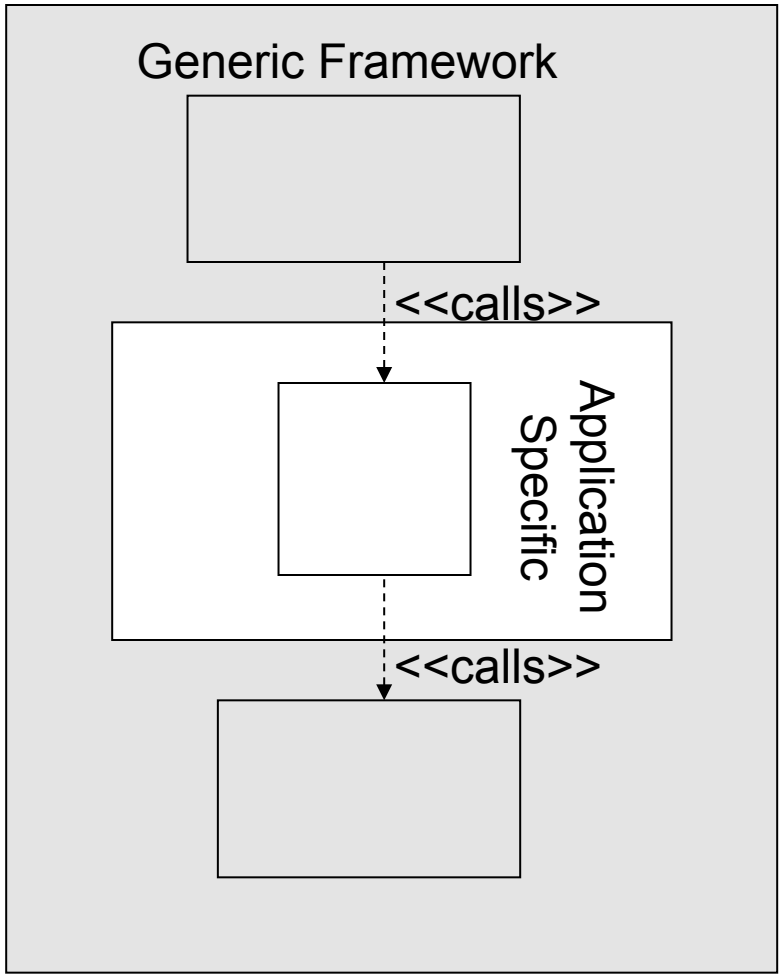
a. High-level modules should not depend on low-level modules. Both should depend on abstractions.
b. Abstractions should not depend on details. Details should depend on abstractions.
   - Whereas (package) dependence usually follows the direction of the calls, the DIP suggest otherwise

DIP

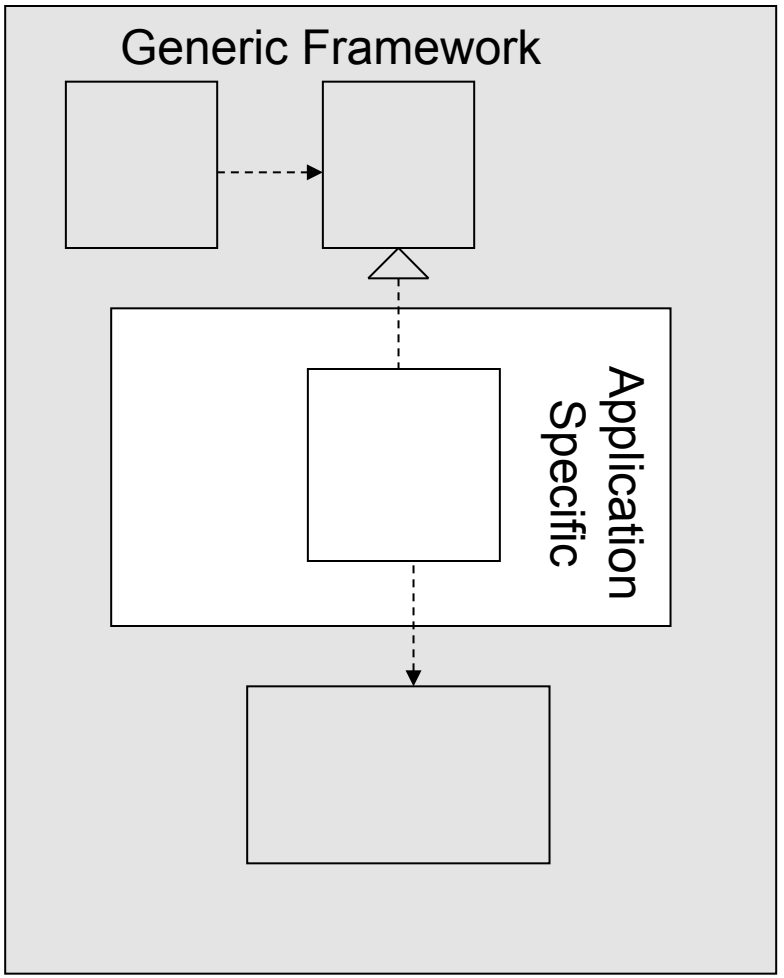


- The DIP is crucial to reusable frameworks.
- The DIP supports the OCP by making the client open to changes of the server
- The DIP requires the LSP so that the client can depend on properties of the server without depending on a specific server.

DIP and Frameworks



Object Structure for a Framework



Class Structure for a Framework

Specifications

- Contracts for methods
- Contracts for objects
  - Clear box. Contracts are in terms of the actual state of the objects
    - Inheritance: LSP Preconditions may be weakened. Postconditions may be strengthened.
  - Black box. Contracts are in terms of abstract (model) states.
- Automated verification
  - Modern tools can verify respect for contracts.

The End

Go forth and create marvels