NITTE
EDUCATION TRUST

Agile Software Design

Dr.Vani Vasudevan

NITTE
EDUCATION TRUST

Outline

- Principles of clean code and design
- SOLID principles and their applications in Agile development
- Refactoring techniques and tools
- Design patterns in Agile software development
- Agile architecture and design considerations

NITTE
EDUCATION TRUST

Principles of clean code and design

- Agile methods have an approach to software design that includes
  - identifying aspects of bad design
  - avoiding those aspects via a set of principles

NITTE
EDUCATION TRUST

Bad Design

- Rigidity: The design is hard to change
- Fragility: The design is easy to break
- Immobility: The design is hard to reuse
- Viscosity: It is hard to do the right thing
- Needless Complexity: Overdesign
- Needless Repetition: Copy and Paste
- Opacity: Disorganized Expression

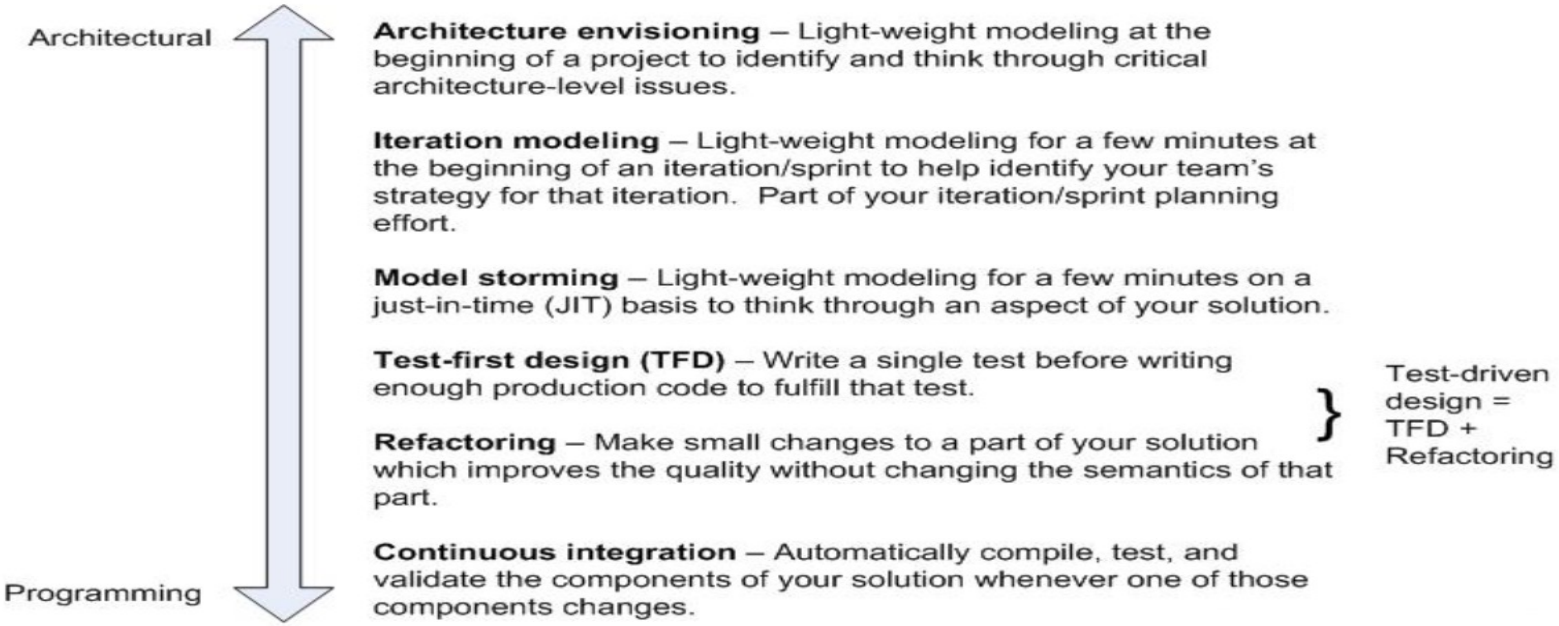
NITTE
EDUCATION TRUST

Good Design Principles

- The Single Responsibility Principle
- The Open-Closed Principle
- The Liskov Substitution Principle
- The Interface Segregation Principle
- The Dependency Inversion Principle

NITTE
EDUCATION TRUST

Agile design practices



NITTE
EDUCATION TRUST

Role of design Principles

- Agile develops software in small increments.
- Client requests evolve with development.
- Frequent deliveries ensure better quality.
- More deliveries lead to higher quality.
- Changes are welcomed at any stage.
- Change requests show market understanding.
- Code must remain flexible for changes.
- No extra effort for future needs.
- Focus is on the present product.

NITTE EDUCATION TRUST

SOLID design Principles

SOLID is an acronym for the following design principles:

- **S**ingle Responsibility Principle
- **O**pen-Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency-Inversion Principle

SOLID principles are hard to spot in large projects.

NITTE
EDUCATION TRUST

Single Responsibility Principle(SRP)

- SRP defines responsibility as a reason to change.
- Changing requirements alter class responsibilities.
- Multiple responsibilities increase change frequency.
- Coupled responsibilities create fragile design.
- Fragility causes unrelated design breaks.
- Example: A class for phone functionalities.

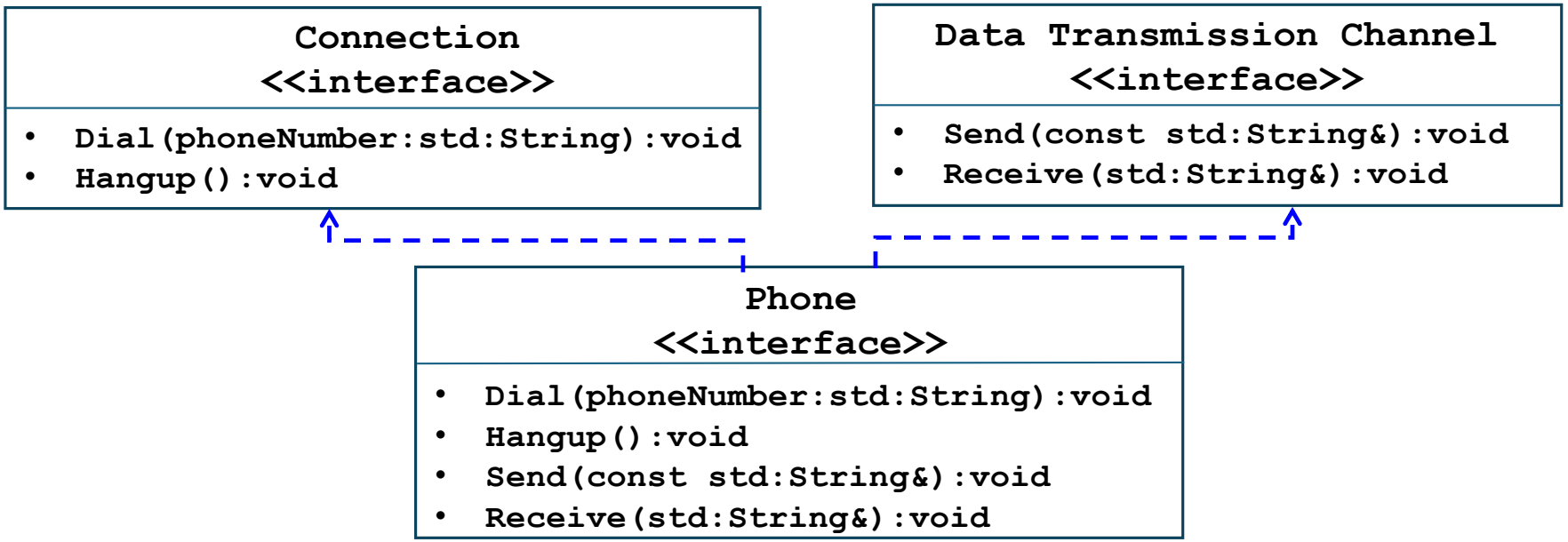
NITTE
EDUCATION TRUST

Single Responsibility Principle

```
class Phone
{
    public void Dial(const string& phoneNumber);
    public void Hangup();
    public void Send(const string& message);
    public Receive(const string& message);
};
```

- Methods represent phone-related functionalities.
- Class has two responsibilities: connection & transmission.
- Changes in connection methods cause rigidity.
- All dependent classes need recompilation.
- Re-design splits responsibilities to prevent issues

Single Responsibility Principle



NITTE
EDUCATION TRUST

Single Responsibility Principle

- Responsibilities are separated to avoid coupling.
- Connection changes won’t affect transmission.
- Separation isn’t needed if no changes occur.
- Split responsibilities only if changes interact.
- SRP is simple but hard to apply.
- Software design relies on finding responsibilities.

NITTE
EDUCATION TRUST

Open-Closed Principle(OCP)

- A class should be open to extension but closed to modification
  - Allows clients to code to class without fear of later changes

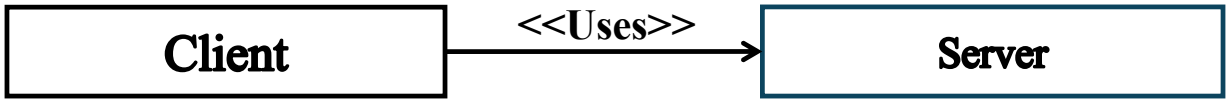
NITTE
EDUCATION TRUST

OCP

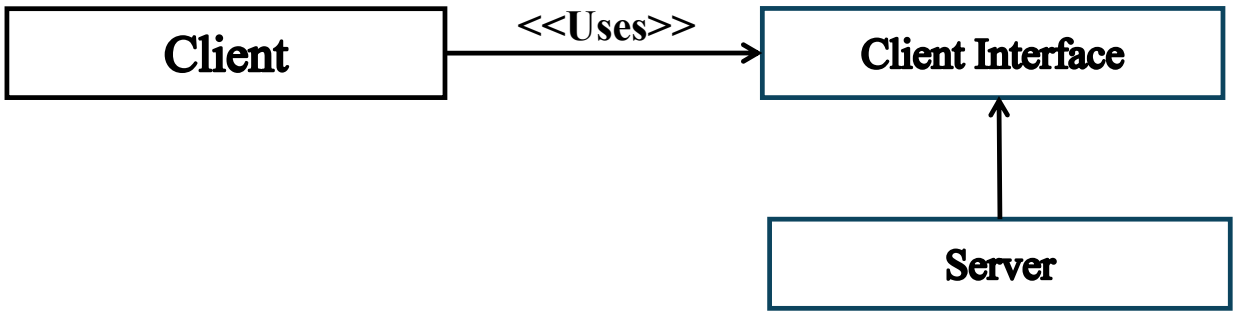
- Rigidity occurs when changes affect many modules.
- OCP promotes refactoring to avoid future changes.
- New features should be added, not modified.
- Open: Code behavior can be extended.
- Closed: Existing code remains unchanged.

NITTE
EDUCATION TRUST

- Abstraction modifies behavior without code changes.
- OOP allows fixed interfaces with many implementations.
- Concrete classes violate the Open-Closed Principle.
- Client class depends directly on Server class.
- Changing the server requires modifying Client class

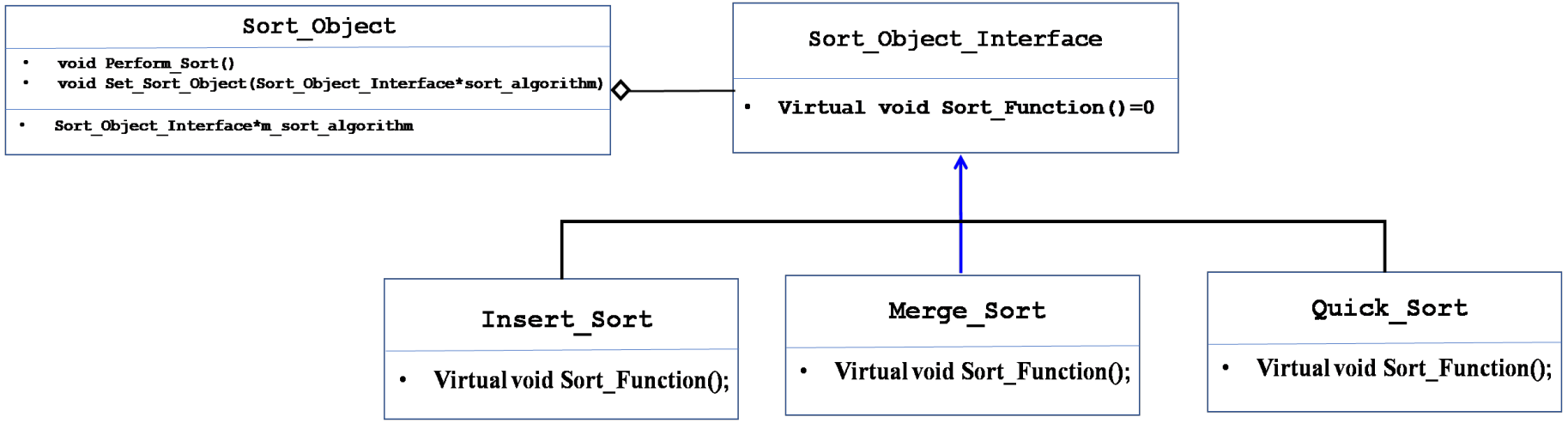


Example which does not comply with the OCP 1 principle



Example observing the OCP principle

- Abstract classes relate more to clients.
- Naming reflects client association, not implementation.
- OCP is used in Strategy and Plugin patterns.
- Figure 4 shows an OCP-compliant design.



Corresponding design, which observes the open-closed principle

- Sort_Object class handles object sorting.
- Sorting logic is in Sort_Object_Interface.
- Derived classes must implement Sort_Function().
- They can customize the sorting implementation.
- Behavior extends by creating new subtypes.

NITTE
EDUCATION TRUST

Highlights of OCP

- OCP relies on abstraction and polymorphism.
- New functionality should use abstractions.
- Abstractions allow future behavior extensions.
- Not always needed, useful for frequent changes.
- OCP compliance increases development effort.
- Abstractions add complexity to software design.
- OCP enhances flexibility, reusability, maintainability.

NITTE
EDUCATION TRUST

Liskov Substitution Principle (LSP)

- Subclasses need to respect the behaviors defined by their superclasses
  - if they do, they can be used in any method that was expecting the superclass
- If a superclass method defines pre and post conditions
  - a subclass can only weaken the superclass pre-condition, and can only strenghen the superclass post-condition

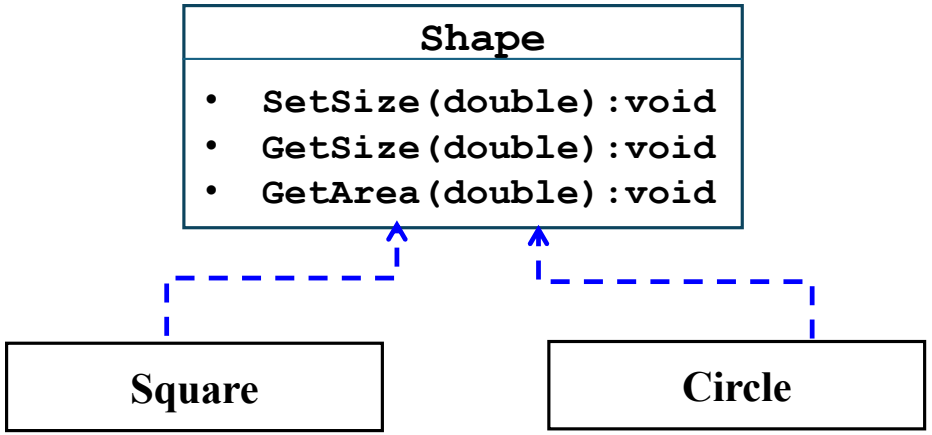
NITTE
EDUCATION TRUST

LSP

- LSP relies on inheritance for abstraction.
- Derived classes must extend, not replace, functionality.
- Base class pointers should work with derived classes.
- Violating LSP causes unexpected program behavior.
- LSP violations become clear when issues arise.
- Example: Shape class with SetSize method.

```
class Shape
{
        public:
                void SetSize(double size);
                void GetSize(double& size);
        private: double mSize;
};
```

- The application can be extended by adding the Square and Circle classes.
- Taking into consideration the fact that the inheritance models an IS_A relationship, the new Square and Circle classes can be derived from the Shape class.



The LSP principle is a mere extension of the Open-Closed principle and it means that, when we add a new class derived in an inheritance hierarchy, we must make sure that the newly added class extends the behaviour of the base class, without modifying it.

NITTE
EDUCATION TRUST

Example (I)

```
class Rectangle {
   private int width;
   private int height;
   public void setHeight(int h) { height = h }
   public void setWidth(int w) { width = w }
}
```

- postcondition of setHeight?
- postcondition of setWidth?

NITTE
EDUCATION TRUST

Example (II)

```
class Square extends Rectangle {
   public void setHeight(int h) {
      super.setHeight(h)
      super.setWidth(h)
   }
   public void setWidth(int w) {
      super.setWidth(w)
      super.setHeight(w)
   }
}
```

Does this maintain the postconditions of the superclass?

NITTE
EDUCATION TRUST

Example (III)

- Let's check

```
public void checkArea(Rectangle r) {
    r.setWidth(5);
    r.setHeight(4);
    assert (r.area() == 20);
    // fails when Square is passed
}
```

- Whoops!

NITTE
EDUCATION TRUST

Example (IV)

- In order to achieve the Liskov substitutability principle, Square's methods needed to have equal or stronger postconditions than its superclass
  - For setWidth:
    - Rectangle: width == w && height == old.height
    - Square: width == w && height == w
  - The postcondition for Square is weaker than the postcondition for Rectangle because it does not attempt to enforce the clause (height == old.height)

NITTE
EDUCATION TRUST

Interface Segregation Principle (ISP)

- Clients should not be forced to depend on methods that they do not use
- Concerns classes with “fat” interfaces
  - what we’ve been calling a non-cohesive module
- A class with a “fat” interface has groups of methods that each service different clients
  - This coupling is bad however, since a change to one group of methods may impact the clients of some other group

NITTE
EDUCATION TRUST

Example (I)

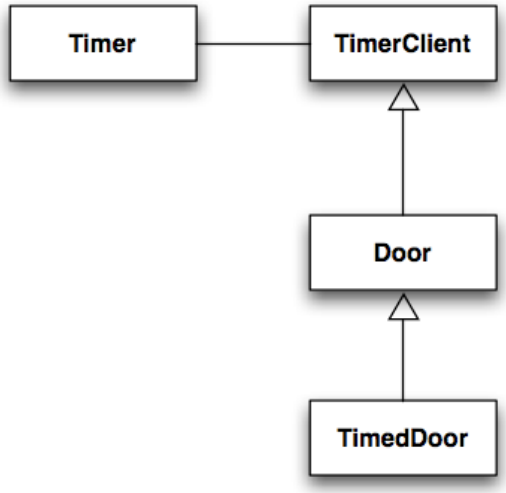
- Security System with Doors that can be locked and unlocked
  - One type of Door is a TimedDoor that needs to sound an alarm if its left open too long
  - Timing is handled in the system via a Timer class that requires its clients to implement an interface called TimerClient
  - TimedDoor needs to make use of Door and TimerClient

NITTE
EDUCATION TRUST

Example (II)

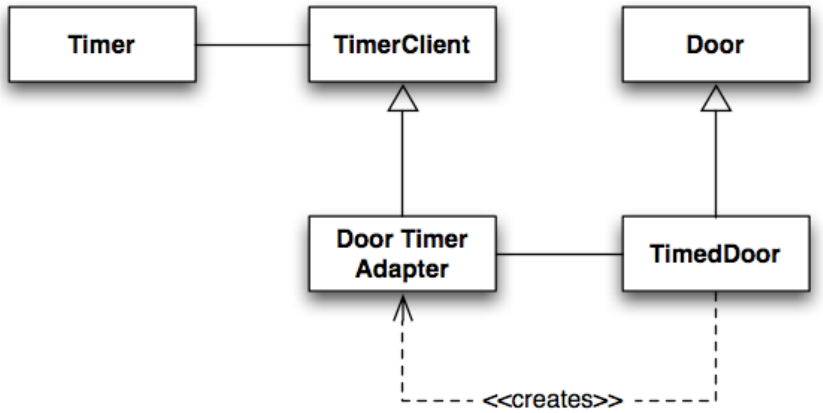
- A naïve solution would be to have the Door interface extend the TimerClient interface
  - This would allow TimedDoor to have access to the routines in TimerClient that would then allow it to interact with the Timer class
- Unfortunately, this “pollutes” the interface of Door with methods that only one of its subclasses needs
  - All other door types get no benefit from the TimerClient methods contained in the Door interface

Example (III)



Here, Door is made to extend TimerClient so that TimedDoor can interact with the Timer object. Unfortunately, now all subclasses of Door depend on TimerClient even if they don't need Timing functionality!

Example (IV)



The solution is to keep the interfaces separate and make use of the adapter pattern to allow TimedDoor access to Timer and its functionality; note, here that this solution keeps TimedDoor independent of Timer and TimerClient

NITTE
EDUCATION TRUST

Interface Segregation Principle (ISP)

- Clients shouldn’t depend on unused methods.
- Interfaces should include only client-specific methods.
- Unnecessary methods force redundant implementations.
- Example: Robot Employee shouldn’t implement Eat().
- Interfaces with irrelevant methods are “polluted.”

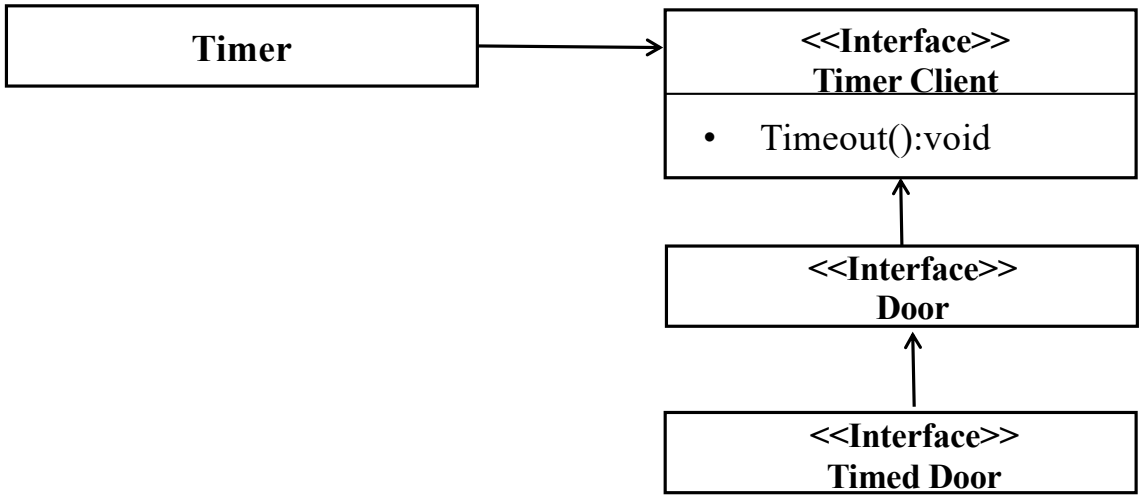
```
class Timer
{
          public:
               void Register(int timeout, TimerClient* client);
};

class TimerClient
{
         public:
               virtual void TimeOut();
};

class Door
{
         public:
               virtual void Lock() = 0;
               virtual void Unlock() = 0;
               virtual bool IsDoorOpen() = 0;
};
```

NITTE
EDUCATION TRUST

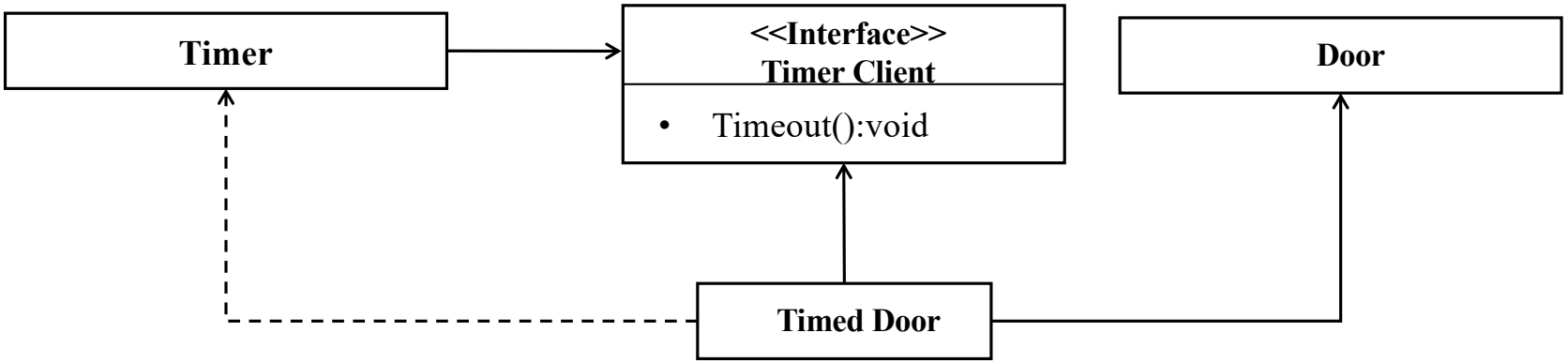
Interface Segregation Principle (ISP)



Class diagram containing: the TimerClient interface, the Door interface and the TimedDoor class

NITTE
EDUCATION TRUST

Interface Segregation Principle (ISP)



Mechanism of multiple inheritance with the Interface Segregation principle in design

NITTE
EDUCATION TRUST

Dependency-Inversion Principle

- High-level modules should not depend on low-level modules; Both should depend on abstractions

- In response to structured analysis and design, in which stepwise refinement leads to the opposite situation
  - high-level modules depend on lower-level modules to get their work done

NITTE
EDUCATION TRUST

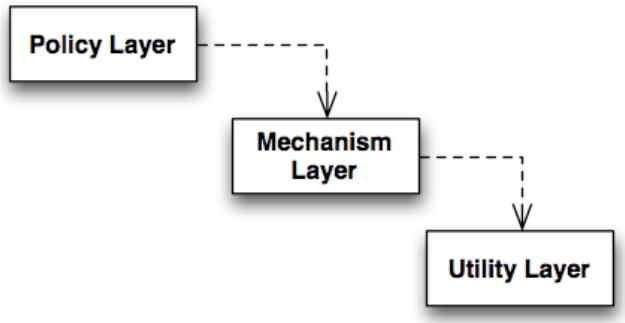
Why “inversion”?

- DIP attempts to “invert” the dependencies that result from a structured analysis and design approach
  - High-Level modules tend to contain important policy decisions and business rules related to an application; they contain the “identity” of an application
    - If they depend on low-level modules, changes in those modules can force the high-level modules to change!
    - High-level modules should not depend on low-level modules in any way

NITTE
EDUCATION TRUST

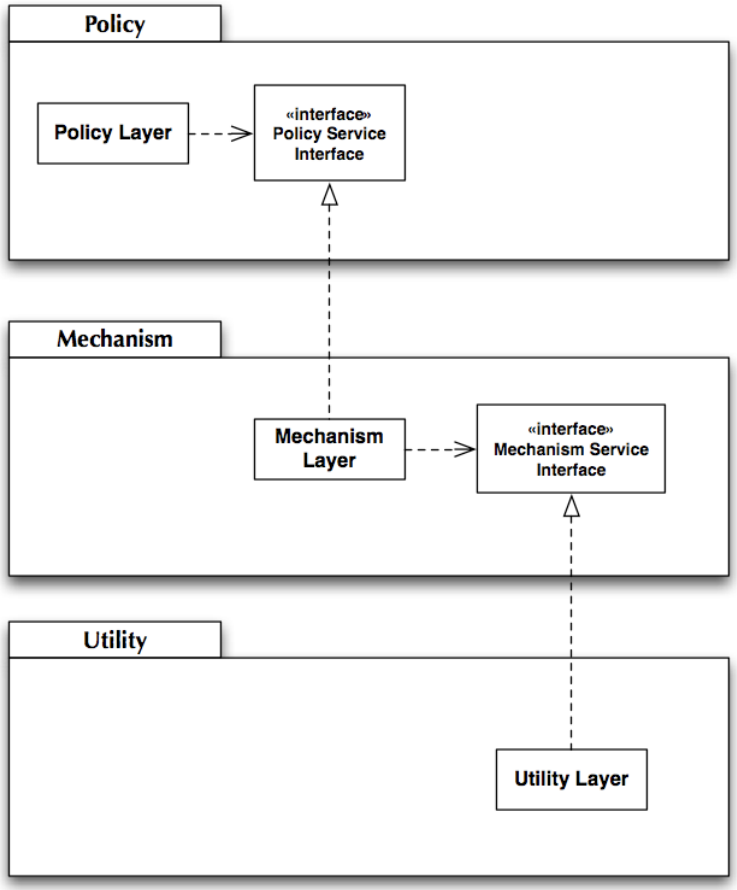
Solution (I)

- In a layered system, a “higher” layer defines an interface that lower layers implement
- Typical Layered System



Solution (II)

Layered System
with DIP applied



NITTE
EDUCATION TRUST

Inversion of Ownership

- Its not just an inversion of dependency, DIP also inverts ownership
  - Typically a service interface is “owned” or declared by the server, here the client is specifying what they want from the server

NITTE
EDUCATION TRUST

Depend on Abstraction

- A simple heuristic based on DIP
  - No variable should reference a concrete class
  - No class should derive from a concrete class
  - No method should override an implemented method of any of its base classes

- Note: this is just a heuristic, not something that can be universally applied
  - but its use can lead to lower coupling within a system

NITTE
EDUCATION TRUST

Refactoring

"Process of restructuring existing code without changing its external functional behaviour to improve nonfunctional attributes of the software" is generally known as **Refactoring**.

NITTE
EDUCATION TRUST

Refactoring

- Agile starts with basic requirements.
- Functionality evolves with learning and changes.
- Code changes over time for fixes and improvements.
- Growing complexity makes code harder to understand.
- Duplicate code arises due to time constraints.
- Refactoring helps debugging and adding features.

Reasons why Refactoring is Important:

- To improve the design of software/application.
- To make software easier to understand.
- To find bugs
- To make program run faster.
- To fix existing legacy database
- To support revolutionary development
- To provide greater consistency for user.

Reasons why Refactoring is Important:

- Refactoring improves software consistency.
- It applies to code, design, and UI.
- Bad code spreads if not refactored.
- Enhancing existing UI simplifies future changes.
- Better design makes code easier to work with.
- Refactoring aids maintainability and extension.
- It enables iterative, incremental code evolution.

NITTE
EDUCATION TRUST

Refactoring Techniques

- Refactoring increases code abstraction.
- It separates code into logical parts.
- Improves naming of variables, classes, and functions.
- Optimizes code block arrangement.

NITTE
EDUCATION TRUST

Refactoring Technique Examples

The most frequently used **refactoring technique examples** are presented below:

Encapsulate Field: Use getters and setters instead of direct access.

Extract Class: Move responsibilities into new classes for SRP compliance.

Extract Interface: Separate shared interface parts for different clients.

Extract Local Variable: Improve readability and traceability of expressions.

Extract Method: Split long methods into smaller, self-contained parts.

NITTE
EDUCATION TRUST

Generalize Type: Move child methods to parents or use interfaces.

Inline: Merge simple code into one method for conciseness.

Introduce Factory: Use factory methods instead of direct constructors.

Introduce Parameter: Convert parameters into named ones for clarity.

Pull Up: Move duplicated code into a superclass.

Push Down: Move rarely used superclass behavior to subclasses.

Replace Conditionals with Polymorphism: Remove redundant conditionals.

NITTE
EDUCATION TRUST

Summary

- Agile design is a process, not an event
  - It’s the continuous application of principles, patterns, and practices to improve the structure and readability of software
- Agile Methods (of which XP is one) are a response to traditional software engineering practices
  - They deemphasize documents/processes and instead value people and communication