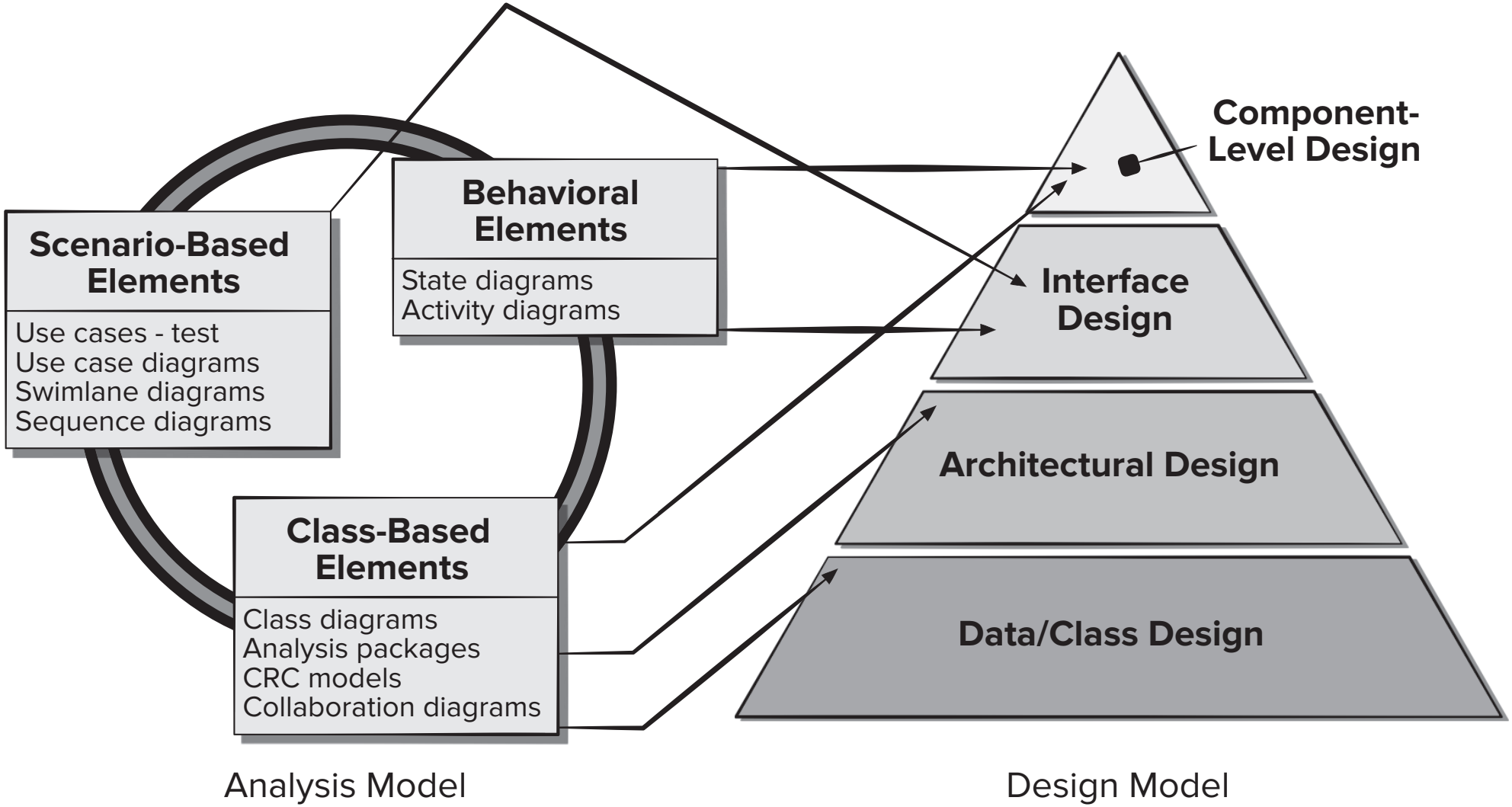
SOFTWARE ENGINEERING

Software Design
Component-Level Design

Course ID 06016410, 06016321

Nont Kanungsukkasem, B.Eng., M.Sc., Ph.D.
nont@it.kmitl.ac.th

Requirements model into Design model



Component-Level Design

Component-Level Design

- Component-level design occurs after the first iteration of architectural design has been completed.
- Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each software component.
- The intent is to translate the design model into operational software.
- Component-level design bridges the gap between architectural design and coding.

Basic Design Principles

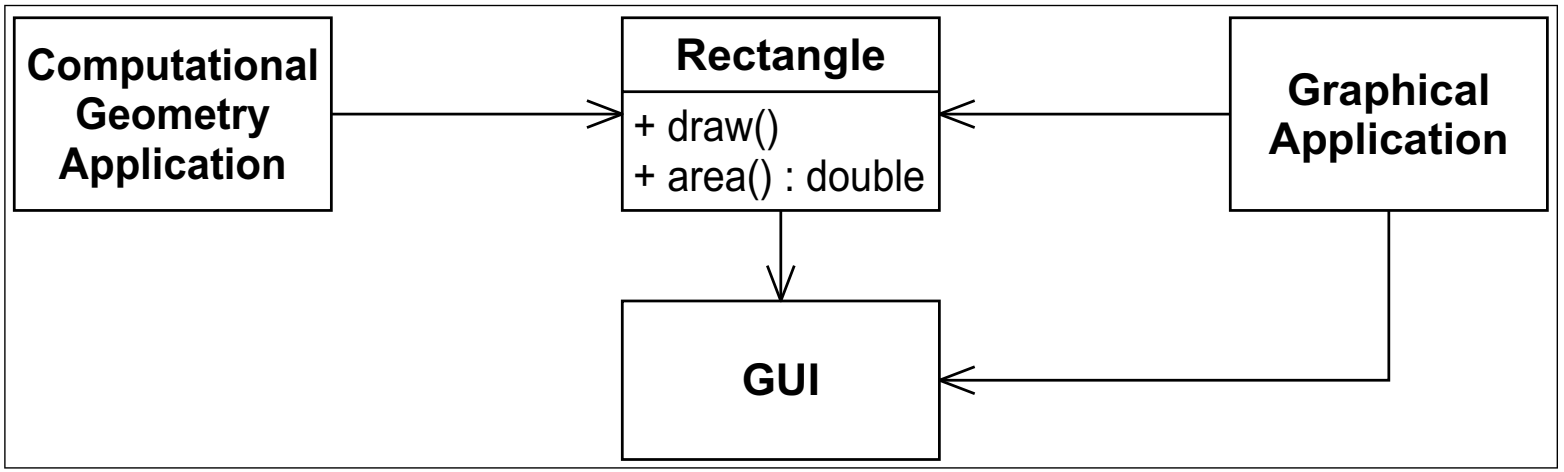
- **SOLID** Design
  - **S** : Single Responsibility Principle
  - **O**: Open-Closed Principle
  - **L** : Liskov Substitution Principle
  - **I** : Interface Segregation Principle
  - **D** : Dependency Inversion Principle

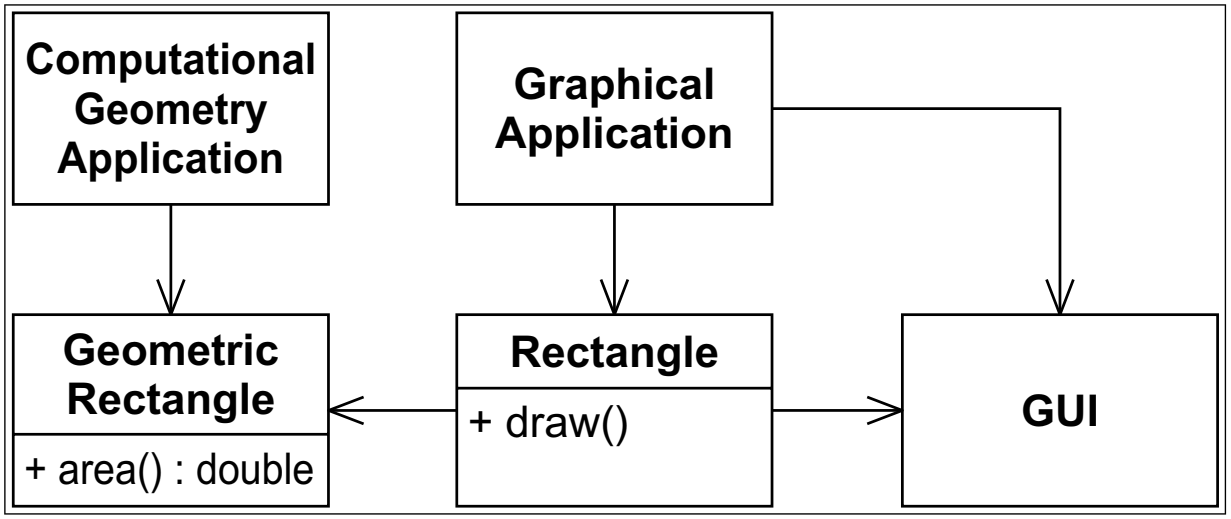
Single Responsibility Principle (SRP)

- A class should only have a single responsibility.
- If a class has more than one responsibility, then the responsibilities become coupled.
- Changes to one responsibility may impair or inhibit the ability of the class to meet the others.
- This kind of coupling leads to fragile designs that break in unexpected ways when changed.
- *Separation of concerns*

> *"A class should have one, and only one, reason to change."*
>
> By Robert C. Martin

One vs. Separated Responsibilities

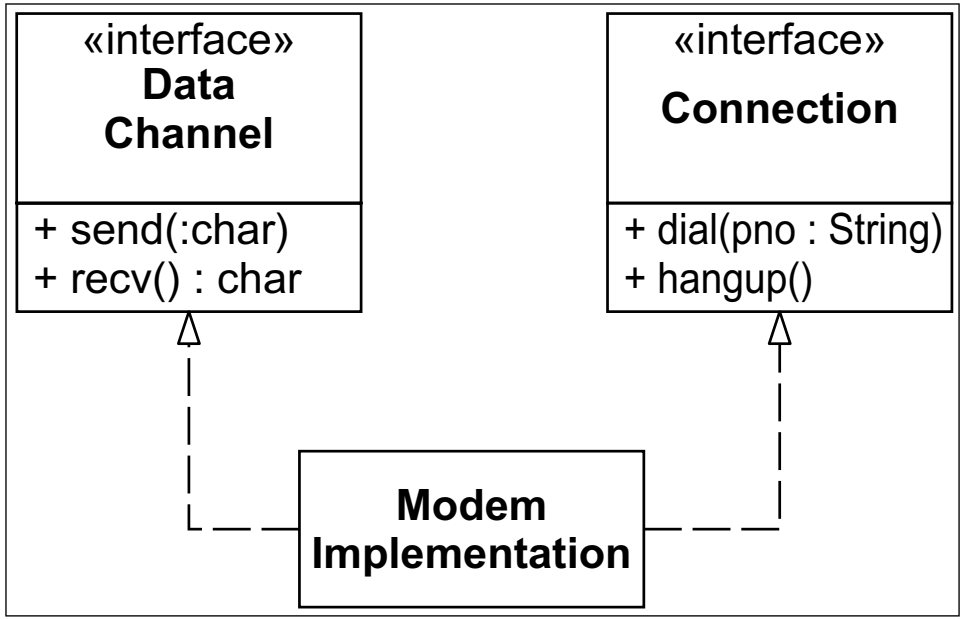




Rigidity vs. Needless Complexity

Modem.java -- SRP Violation

```
interface Modem
{
  public void dial(String pno);
  public void hangup();
  public void send(char c);
  public char recv();
}
```



Example in Python

Example in Python

Comply with SRP

- sry.py

Anti SRP (Violate SRP)

- srp_anti-pattern.py

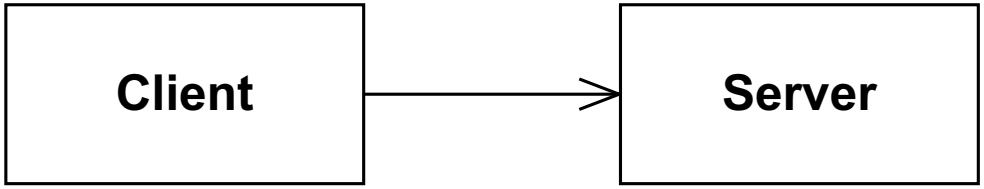
Open-Closed Principle (OCP)

"Software entities (classes, modules, functions, etc.) should be *open* for *extension*, but *closed* for *modification*."

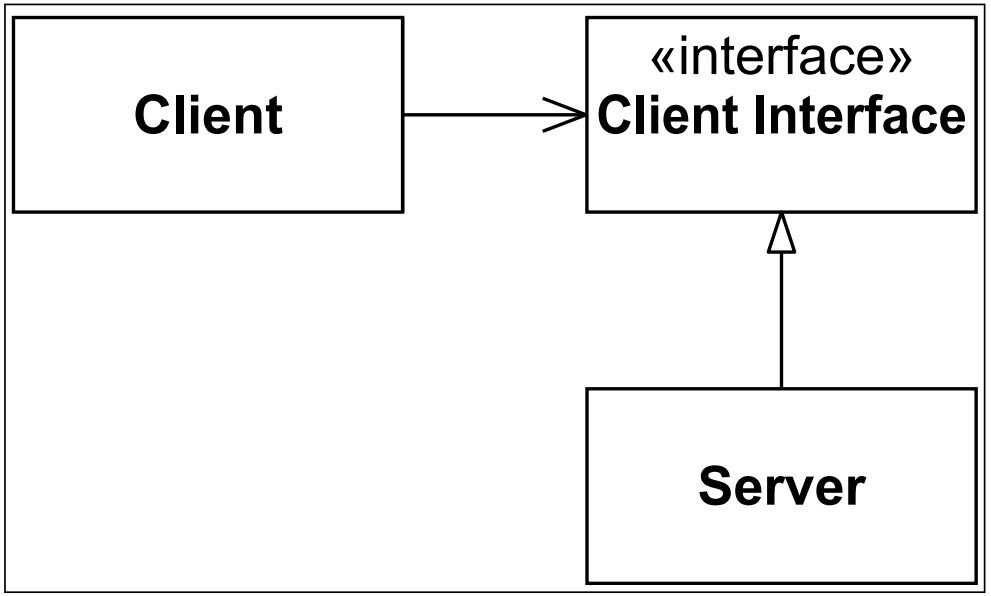
By Robert C. Martin

- "Open for extension."
  - This means that the behavior of the module can be extended.
  - As the requirements of the application change, we are able to extend the module with new behaviors that satisfy those changes.
  - In other words, we are able to change what the module does.
- "Closed for modification."
  - Extending the behavior of a module does not result in changes to the source or binary code of the module.
  - The binary executable version of the module, whether in a linkable library, a DLL, or a Java .jar, remains untouched.

Using Abstraction to Gain Explicit Closure



Client is not open and closed



STRATEGY pattern: Client is both open and closed

```
//--shape.h-----------------------------------------
enum ShapeType {circle, square};
struct Shape
{
    ShapeType itsType;
};

//--circle.h----------------------------------------
struct Circle
{
    ShapeType itsType;
    double itsRadius;
    Point itsCenter;
};
void DrawCircle(struct Circle*);

//--square.h----------------------------------------
struct Square
{
    ShapeType itsType;
    double itsSide;
    Point itsTopLeft;
};
void DrawSquare(struct Square*);

//--drawAllShapes.cc--------------------------------
typedef struct Shape *ShapePointer;
void DrawAllShapes(ShapePointer list[], int n)
{
    int i;
    for (i=0; i<n; i++)
    {
        struct Shape* s = list[i];
        switch (s->itsType)
        {
            case square:
                DrawSquare((struct Square*)s);
                break;
            case circle:
                DrawCircle((struct Circle*)s);
                break;
        } }
}
```

```
class Shape {
  public:
    virtual void Draw() const = 0;
};
class Square : public Shape
{
  public:
    virtual void Draw() const;
};
class Circle : public Shape
{
  public:
    virtual void Draw() const;
};
void DrawAllShapes(vector<Shape*>& list)
{
  vector<Shape*>::iterator i;
  for (i=list.begin(); i != list.end(); i++)
    (*i)->Draw();
}
```

Open-Closed Principle (OCP)

- Anticipation and "Natural" Structure
  - There is no model that is natural to all contexts!
  - guess and construct abstractions -> expensive
- Putting the "Hooks" In
  - Fool Me Once...
    - We take the first bullet, and then we make sure we are protected from any more bullets coming from that gun.
  - Stimulating Change.

Example in Python

Example in Python

Comply with OCP

- ocp.py

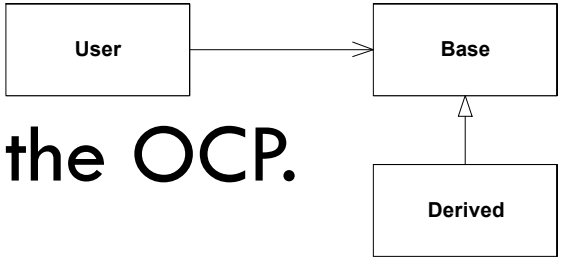
Anti OCP (Violate OCP)

- ocp_anti-pattern.py

Liskov Substitution Principle (LSP)

> *"Subtypes must be substitutable for their base types."*
>
> By Robert C. Martin

- Objects in a program should be able to be replaced with instances of their subtypes without altering the correctness of that program.
- In other words, a component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead.
- The LSP is one of the prime enablers of the OCP.



Example in Python

Example in Python

Anti LSP (Violate LSP)

•lsp_anti-pattern.py

Solution:

• Don’t implement any stricter validation rules on input parameters than implemented by the parent class.

• Apply at the least the same rules to all output parameters as applied by the parent class

IS-A

- IS-A is about Behavior
  - Behaviorally, a Square is not a Rectangle, and it is *behavior* that software is really all about.
  - the IS-A relationship pertains to behavior that can be reasonably assumed.

Design by Contract

- Design by Contract (DbD)
  - the author of a class explicitly states the contract for that class.
  - The contract informs the author of any client code about the behaviors that can be relied on.
  - The contract is specified by declaring preconditions and postconditions for each method.

"A routine redeclaration [in a derivative] may only replace the original *precondition* by one *equal* or *weaker*, and the original *postcondition* by one *equal* or *stronger*"

By Bertrand Meyer

Contracts in Unit Tests

- Specifying Contracts in Unit Tests
  - Contracts can also be specified by writing unit tests.
  - By thoroughly testing the behavior of a class, the unit tests make the behavior of the class clear.
  - Authors of client code will want to review the unit tests so that they know what to reasonably assume about the classes they are using.

Interface Segregation Principle (ISP)

> "Many *client-specific* interfaces are better than One *general-purpose* interface."
>
> By Robert C. Martin

- This principle deals with the disadvantages of "fat" interfaces.
  - Classes that have "fat" interfaces are classes whose interfaces are not cohesive.
  - In other words, the interfaces of the class can be broken up into groups of methods.
  - Each group serves a different set of clients.
  - Thus, some clients use one group of member functions, and other clients use the other groups.
- Clients should not be forced to depend on methods that they do not use.

Example in Python

Example in Python

Comply with ISP

- isp.py

Anti ISP (Violate ISP)

- isp_anti-pattern.py

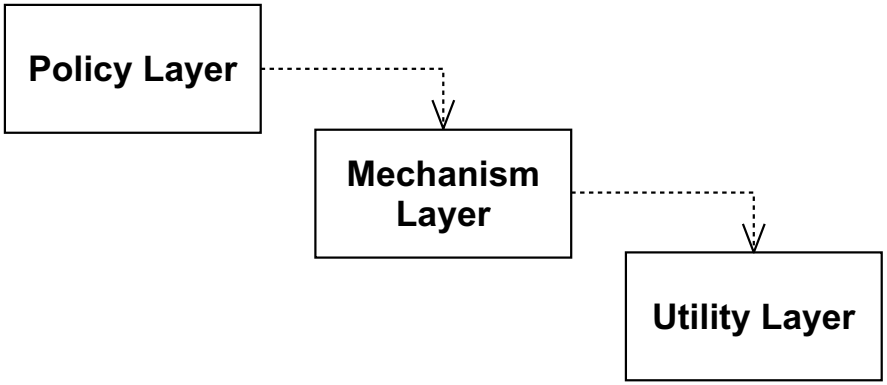
Dependency Inversion Principle (DIP)

"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."

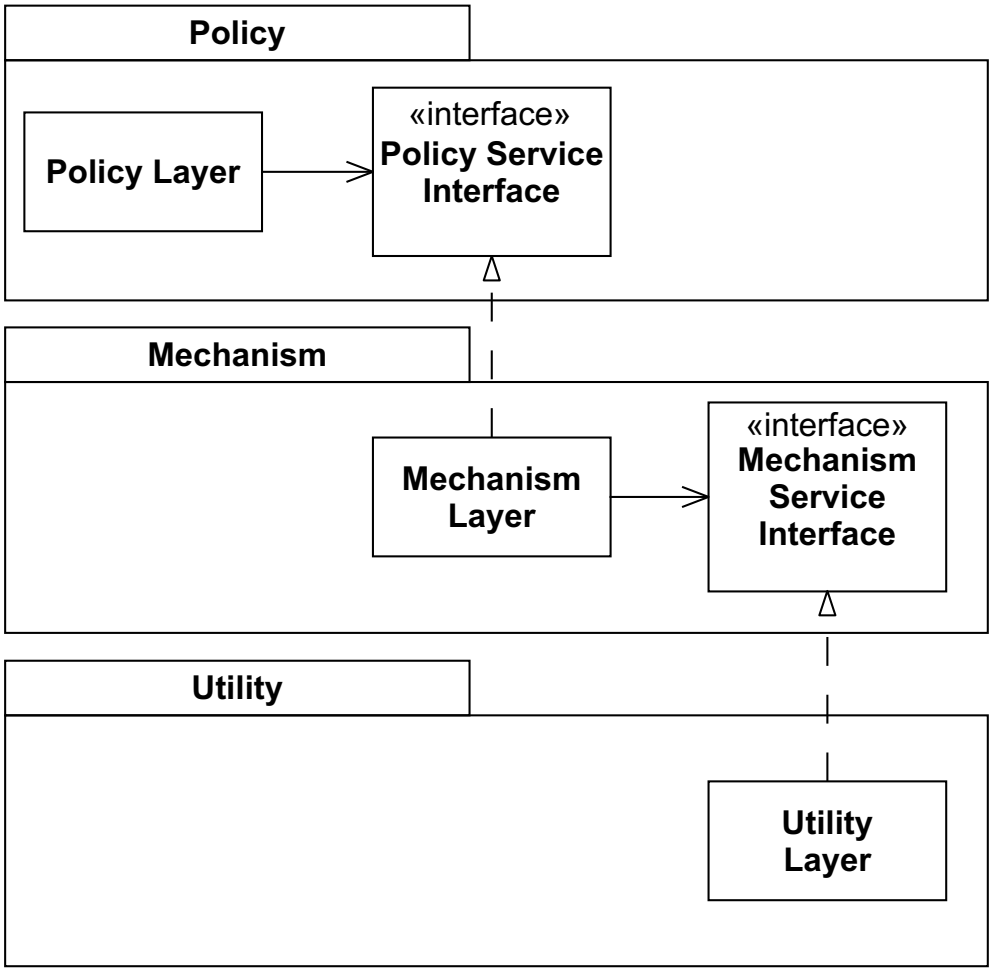
By Robert C. Martin

- Depend on abstractions.
- Do not depend on concretions.

Dependency Inversion Principle (DIP)



Naive layering scheme



Inverted Layers

Example in Python

Example in Python

Comply with DIP

•dip.py

Anti DIP (Violate DIP)

•dip_anti-pattern.py