Analysis, Design, and Software Architecture (BDSA)

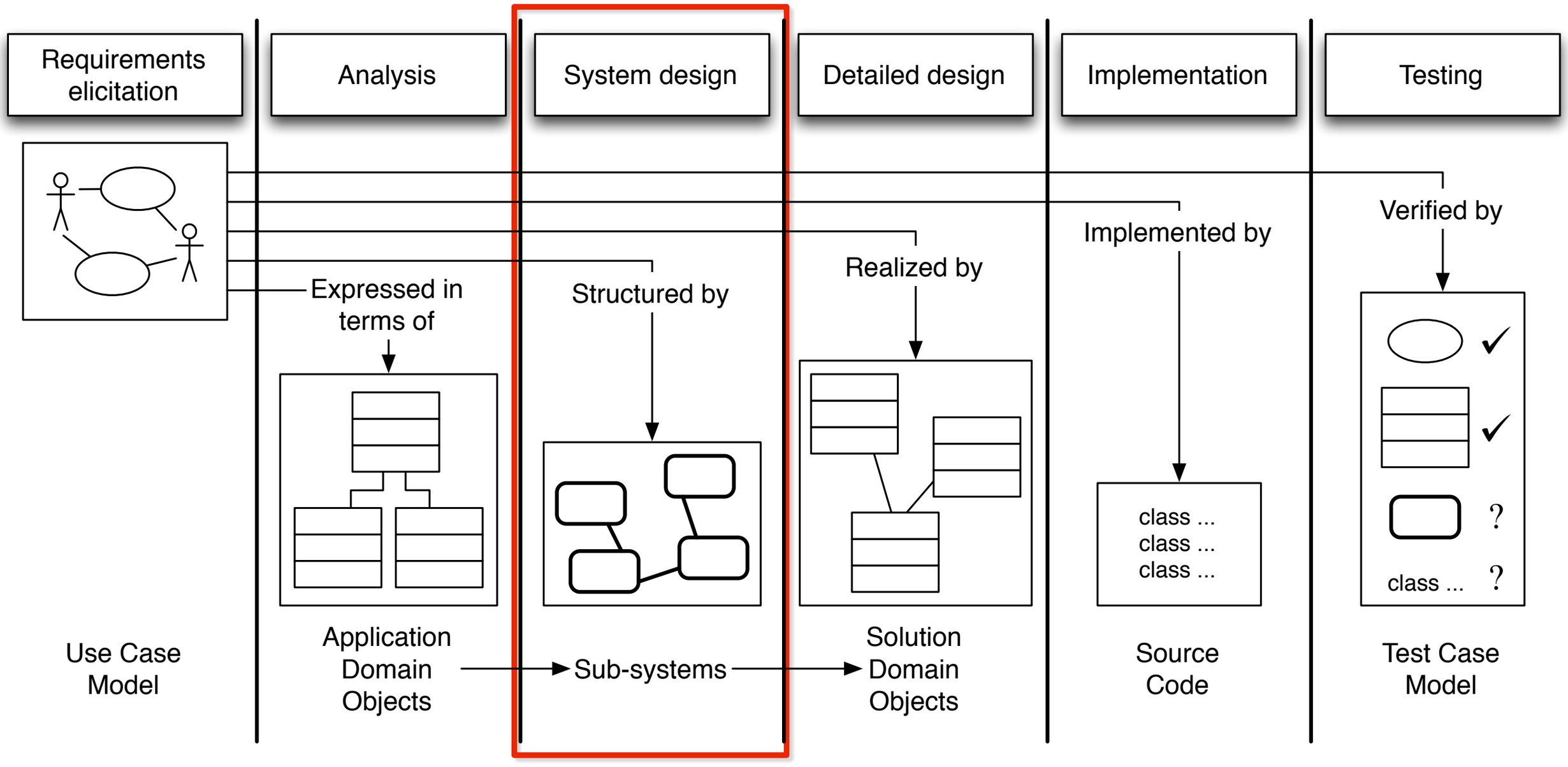
Paolo Tell

SOLID principles

IT University
of Copenhagen

Recap

Software lifecycle activities



Requirements Analysis Document

1. Introduction
   - 1.1 Purpose of the system
   - 1.2 Scope of the system
   - 1.3 Objectives and success criteria of the project
   - 1.4 Definitions, acronyms, and abbreviations
   - 1.5 References
   - 1.6 Overview
2. Current system
3. Proposed system
   - 3.1 Overview
   - 3.2 Functional requirements
   - 3.3 Nonfunctional requirements
     - 3.3.1 Usability
     - 3.3.2 Reliability
     - 3.3.3 Performance
     - 3.3.4 Supportability
     - 3.3.5 Implementation
     - 3.3.6 Interface
     - 3.3.7 Packaging
     - 3.3.8 Legal
   - 3.4 System models
     - 3.4.1 Scenarios
     - 3.4.2 Use case model
     - *3.4.3 Object model*
     - *3.4.4 Dynamic model*
     - 3.4.5 User interface—navigational paths and screen mock-ups
4. Glossary

Requirements Analysis Document

1. Introduction
2. Current system
3. Proposed system
   - 3.1 Overview
   - 3.2 Functional requirements
   - 3.3 Nonfunctional requirements
   - 3.4 System models
     - 3.4.1 Scenarios
     - 3.4.2 Use case model
     - 3.4.3 Object model
       - 3.4.3.1 Data dictionary
       - 3.4.3.2 Class diagrams
     - 3.4.4 Dynamic models
     - 3.4.5 User interface—navigational paths and
4. Glossary

Figure 4-16 Outline of the Requirements Analysis Document (RAD). Sections in *italics* are completed during analysis (see next chapter).

Copyright © 2011 Pearson Education, Inc. publishing as Prentice Hall

Outline

- Literature for the next lectures
  - [OOSE] ch. 6, 7, 8
  - (Optional) [SE9] ch. 6, 7

- Topics covered today:
  - SOLID principles

Symptoms of bad software

1.

2.

3.

Symptoms of bad software

- Confusing. A good software should explain what it is doing.

1.

2.

3.

Symptoms of bad software

- Confusing. A good software should explain what it is doing.

1. Rigidity. To make a modification, you need “touch” a lot of other stuff.
2.
3.

Symptoms of bad software

- Confusing. A good software should explain what it is doing.

1. Rigidity. To make a modification, you need “touch” a lot of other stuff.

2. Fragility. You change something somewhere, and it breaks the software in unexpected areas that are entirely unrelated to the change you made.

Symptoms of bad software

- Confusing. A good software should explain what it is doing.

1. Rigidity. To make a modification, you need “touch” a lot of other stuff.

2. Fragility. You change something somewhere, and it breaks the software in unexpected areas that are entirely unrelated to the change you made.

3. Immobility (no-reusability). The software does more than what you need. The desirable parts of the code are so tightly coupled to the undesirable ones that you cannot use the desirable parts somewhere else.

Symptoms of bad software

- Confusing. A goo

1. Rigidity. To mak ther stuff.
2. Fragility. You cha software in unexpected areas made.
3. Immobility (no-re t you need. The desirable parts of rable ones that you cannot use the de

What do these problems have in common?

Symptoms of bad software

- Confusing. A goo

1. Rigidity. To mak ther stuff.
2. Fragility. You cha software in unexpected areas made.
3. Immobility (no-re t you need. The desirable parts of rable ones that you cannot use the de

What do these problems have in common?

Spaghetti code
Coupling
Dependencies

Symptoms of bad software

- Confusing. A goo

1. Rigidity. To mak her stuff.
2. Fragility. You cha software in unexpected areas made.
3. Immobility (no-re you need. The desirable parts of able ones that you cannot use the de

How do you manage your dependencies?

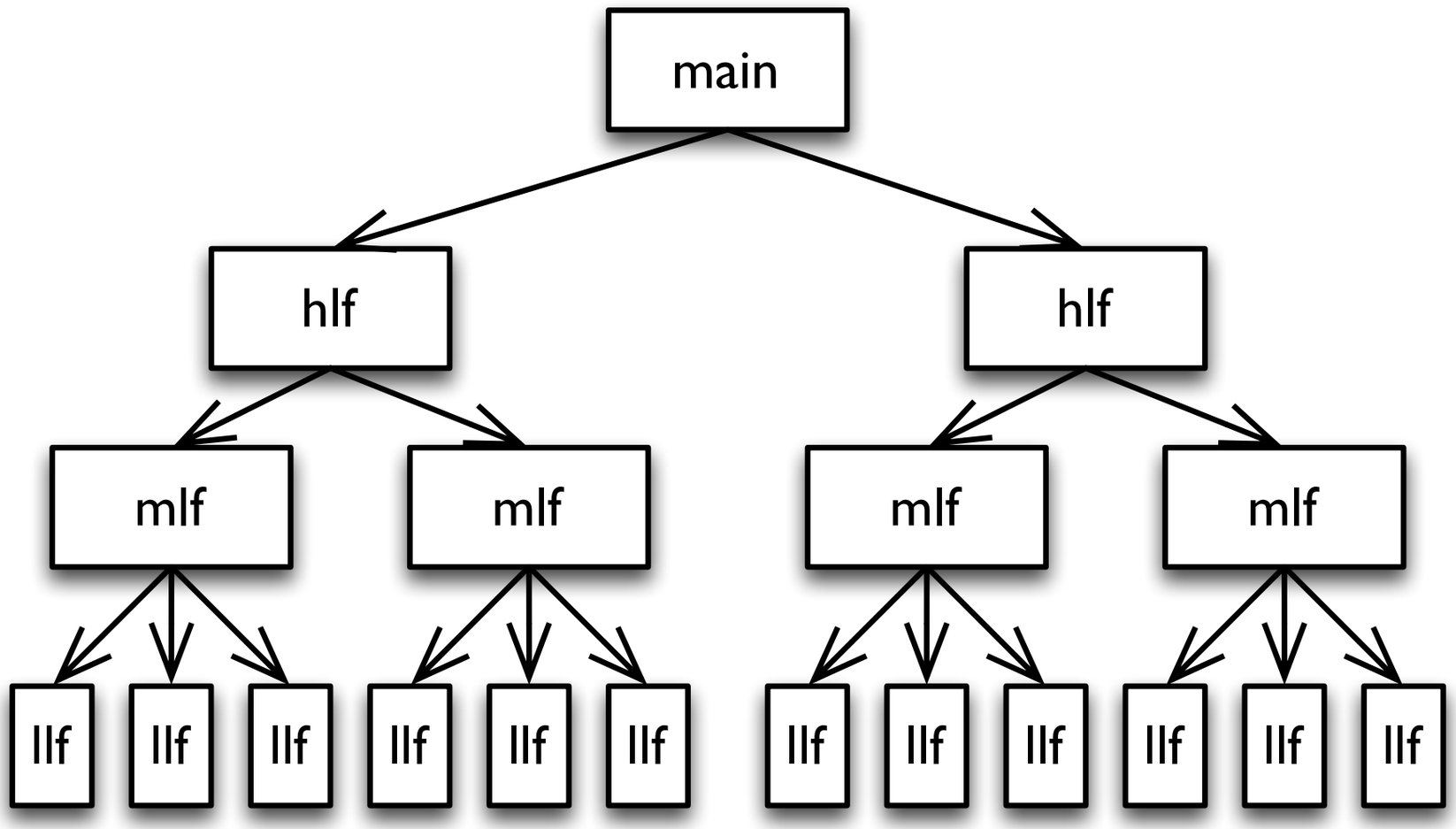
A bit of history

- Simula – 1966 (Ole-Johan Dahl and Kristen Nygaard, Norwegian).
- SmallTalk – late 70s (Alan Kay et al., Xerox PARC).
- C++ – 1985 (Bjarne Stroustrup, Danish).
- Java – 1995.
- C# – 2000.

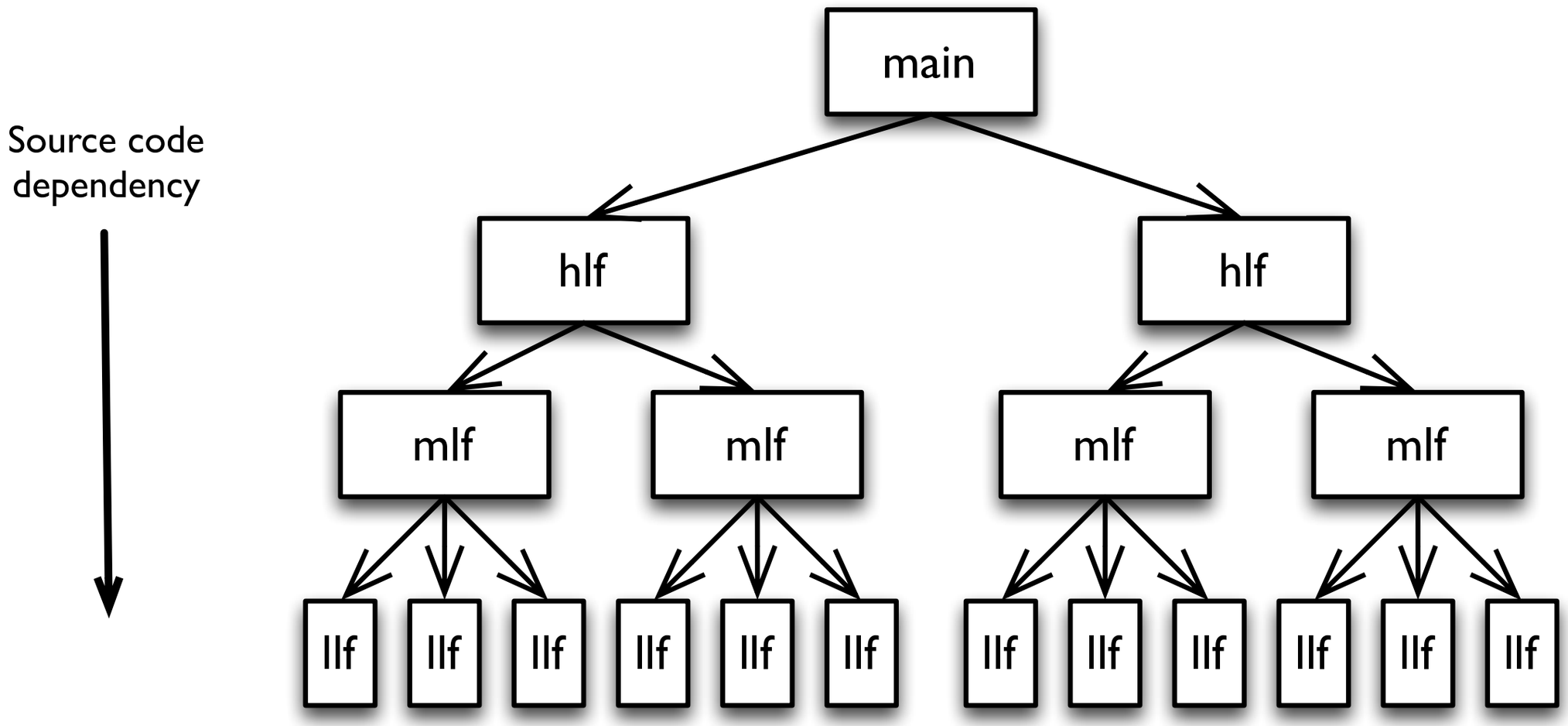
What makes a language OO

- Encapsulation
- Inheritance
- Polymorphism

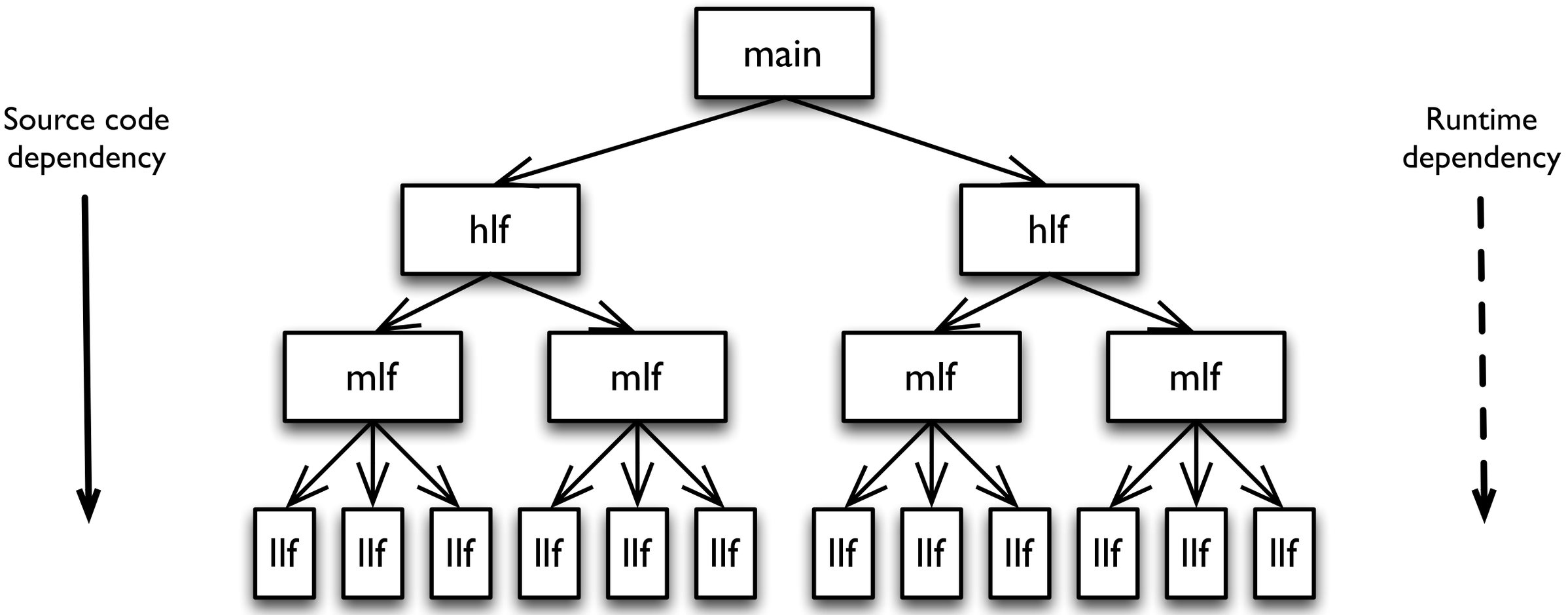
In (procedural) structured programming



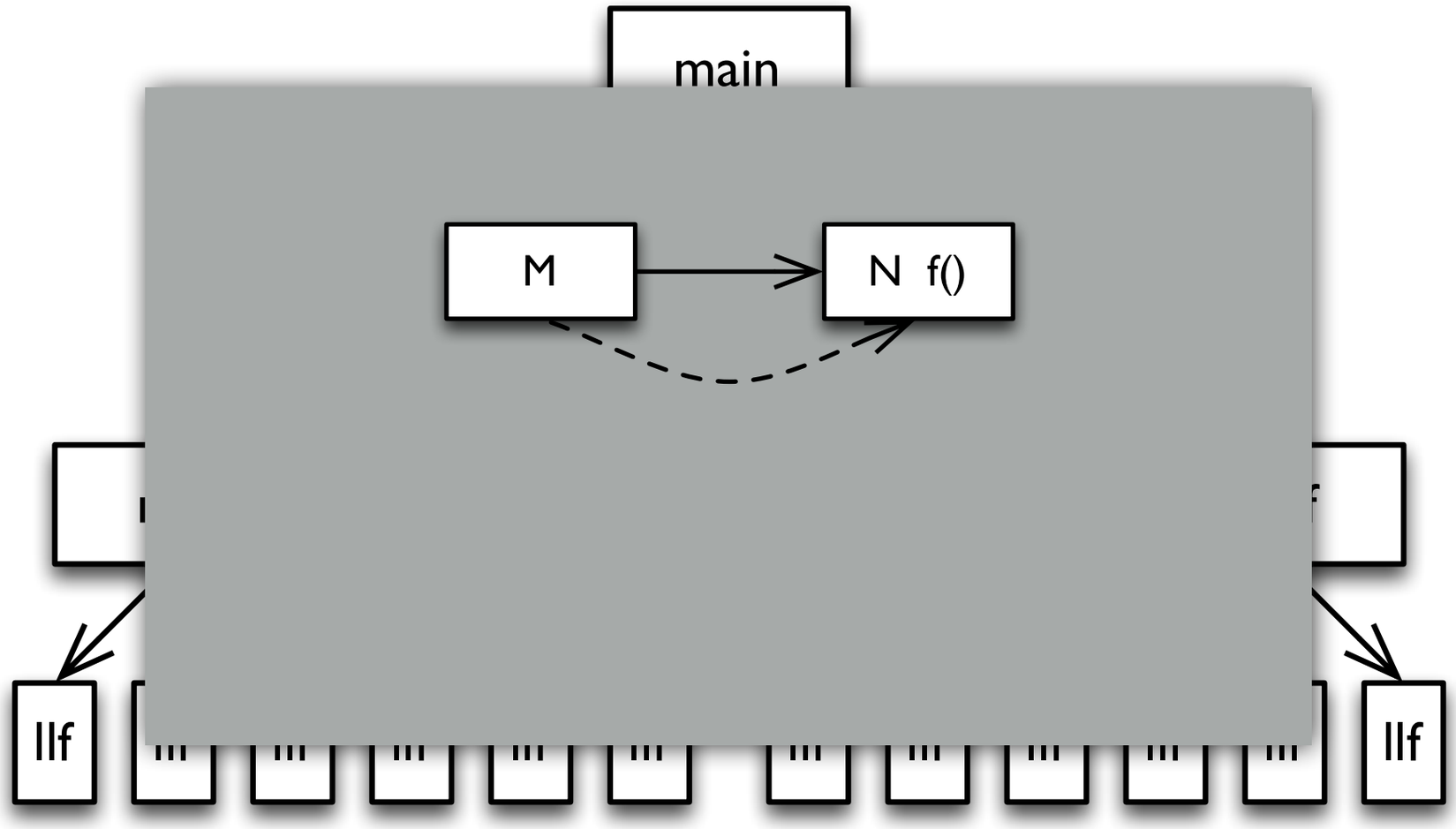
In (procedural) structured programming



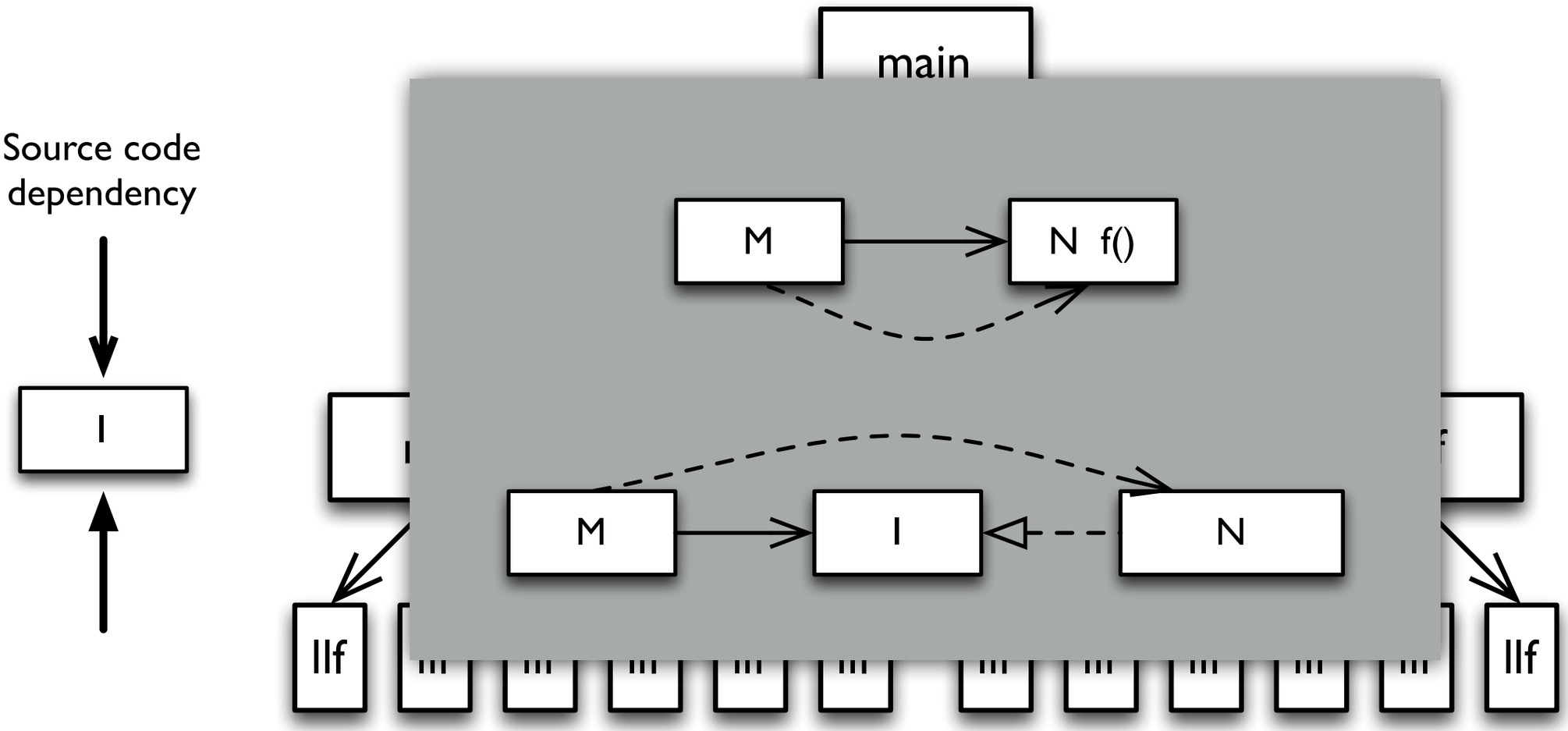
In (procedural) structured programming



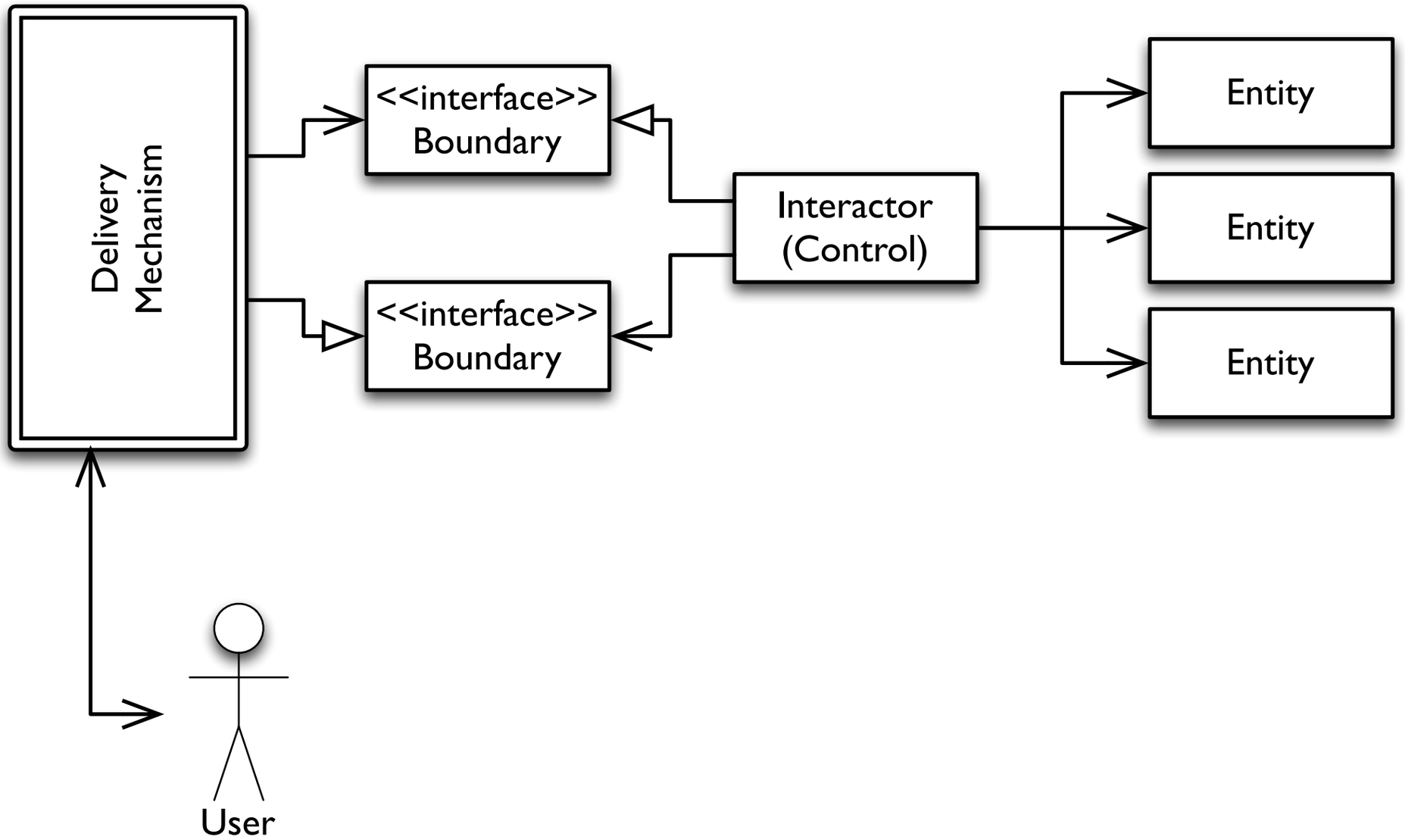
In (procedural) structured programming

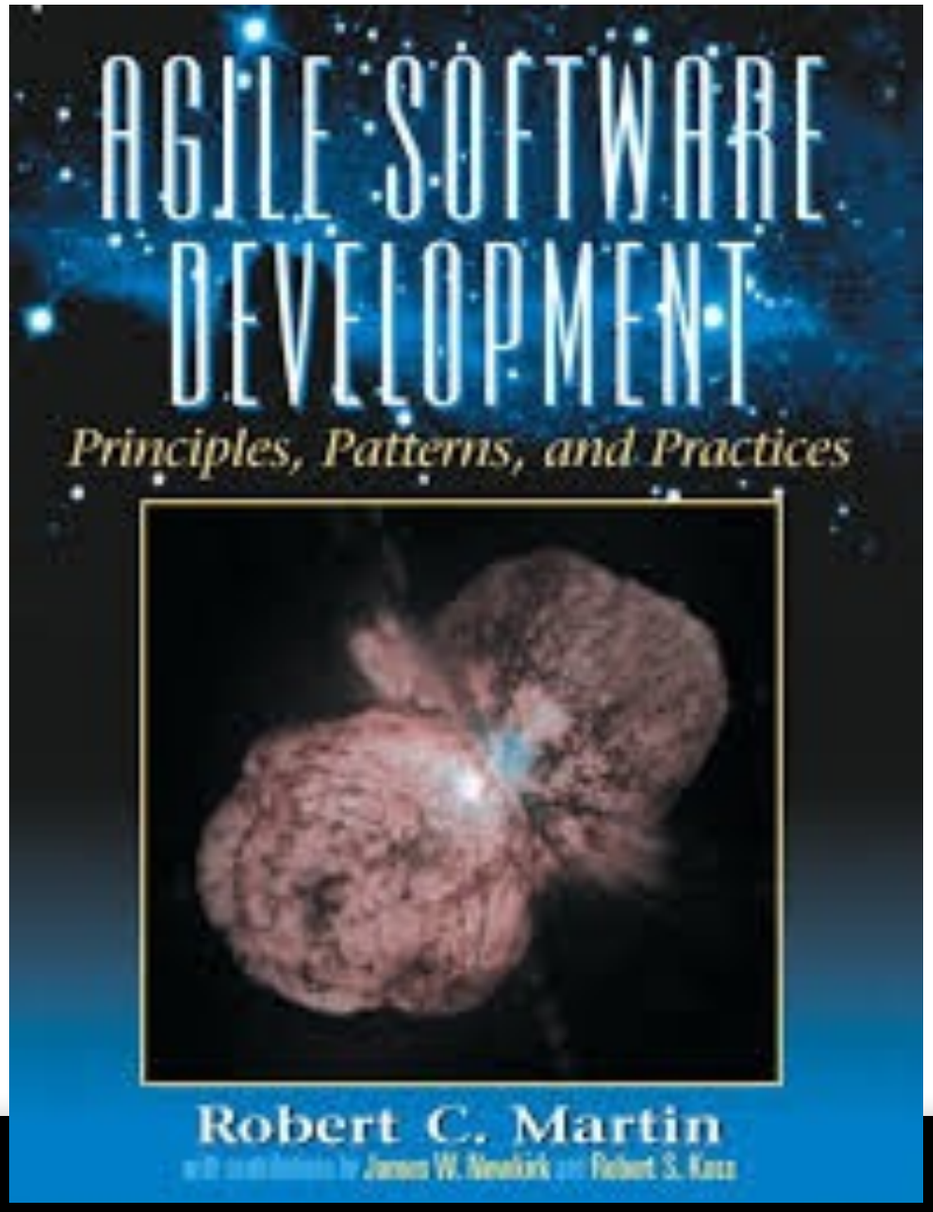


!!! polymorphism !!!



Interactor (Control) - Entity - Boundary




AGILE SOFTWARE DEVELOPMENT
Principles, Patterns, and Practices
Robert C. Martin

S.O.L.I.D.

Robert Cecil Martin (uncle Bob)

S.O.L.I.D.

- Single responsibility principle: "a class should only have one, and only one, reason to change".
- Open/closed principle: "software entities should be open for extensions but closed for modifications". (Bertrand Meyer 1988)
- Liskov substitution principle: "derived classes should be usable through the base class interface, without the need for the user to know the difference". (Barbara Liskov 1987)
- Interface segregation principle: "many client-specific interfaces are better than one general-purpose interface".
- Dependency inversion principle: "depend upon abstractions, do not depend upon concretions".

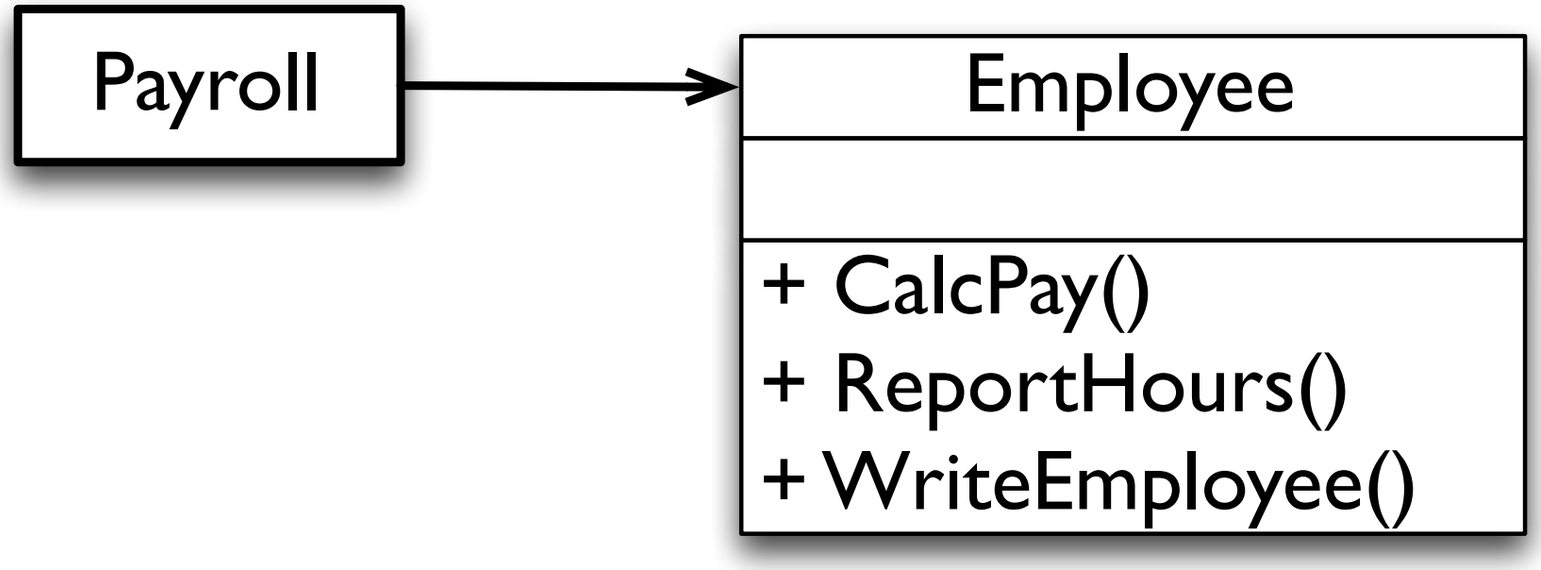
S.O.L.I.D.

- Single responsibility principle: "a class should only have one, and only one, reason to change".
- Open/closed principle: "software entities should be open for extensions but closed for modifications". (Bertrand Meyer 1988)
- Liskov substitution principle: "derived classes should be usable through the base class interface, without the need for the user to know the difference". (Barbara Liskov 1987)
- Interface segregation principle: "many client-specific interfaces are better than one general-purpose interface".
- Dependency inversion principle: "depend upon abstractions, do not depend upon concretions".

SINGLE RESPONSIBILITY PRINCIPLE

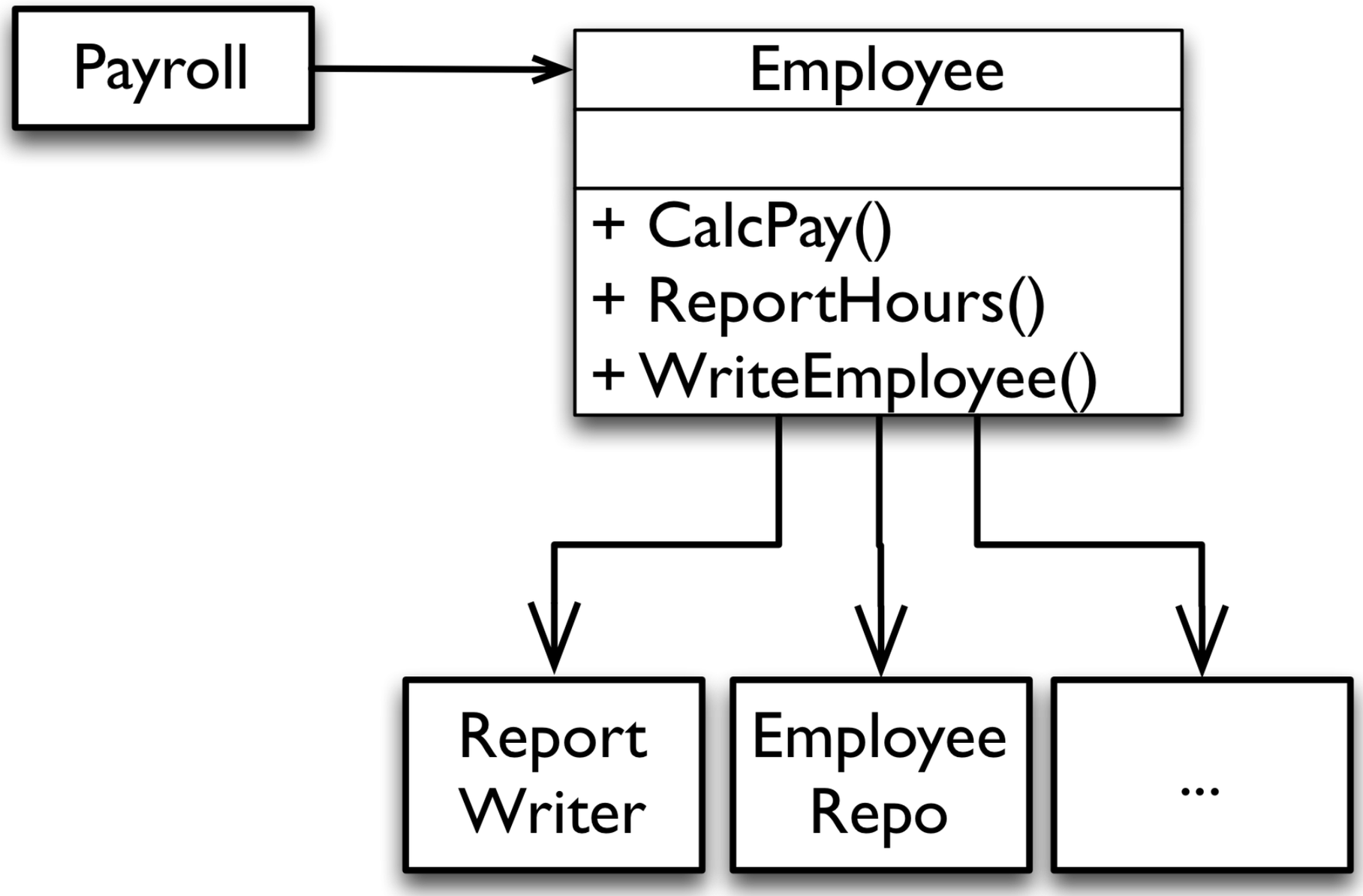
Just Because You Can, Doesn't Mean You Should

Single responsibility principle



- A class should only have one, and only one, reason to change.
- A class should have only a single responsibility.

Single responsibility principle



S.O.L.I.D.

- Single responsibility principle: "a class should only have one, and only one, reason to change".
- Open/closed principle: "software entities should be open for extensions but closed for modifications". (Bertrand Meyer 1988)
- Liskov substitution principle: "derived classes should be usable through the base class interface, without the need for the user to know the difference". (Barbara Liskov 1987)
- Interface segregation principle: "many client-specific interfaces are better than one general-purpose interface".
- Dependency inversion principle: "depend upon abstractions, do not depend upon concretions".

Open/closed principle

Shape.h

```
enum ShapeType {circle, square};
struct Shape
     {enum ShapeType itsType;};
```

Circle.h

```
struct Circle
{
  enum ShapeType itsType;
  double itsRadius;
  Point itsCenter;
};
void DrawCircle(struct Circle*)
```

Square.h

```
struct Square
{
  enum ShapeType itsType;
  double itsSide;
  Point itsTopLeft;
};
void DrawSquare(struct Square*)
```

DrawAllShapes.c

```
#include <Shape.h>
#include <Circle.h>
#include <Square.h>

typedef struct Shape* ShapePtr;

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

- Software entities should be open for extensions but closed for modifications.

Open/closed principle

Shape.h

```
Class Shape
{
public:
   virtual void Draw() const=0;
};
```

Square.h

```
Class Square: public Shape
{
public:
   virtual void Draw() const;
};
```

Circle.h

```
Class Circle: public Shape
{
public:
   virtual void Draw() const;
};
```

DrawAllShapes.cpp

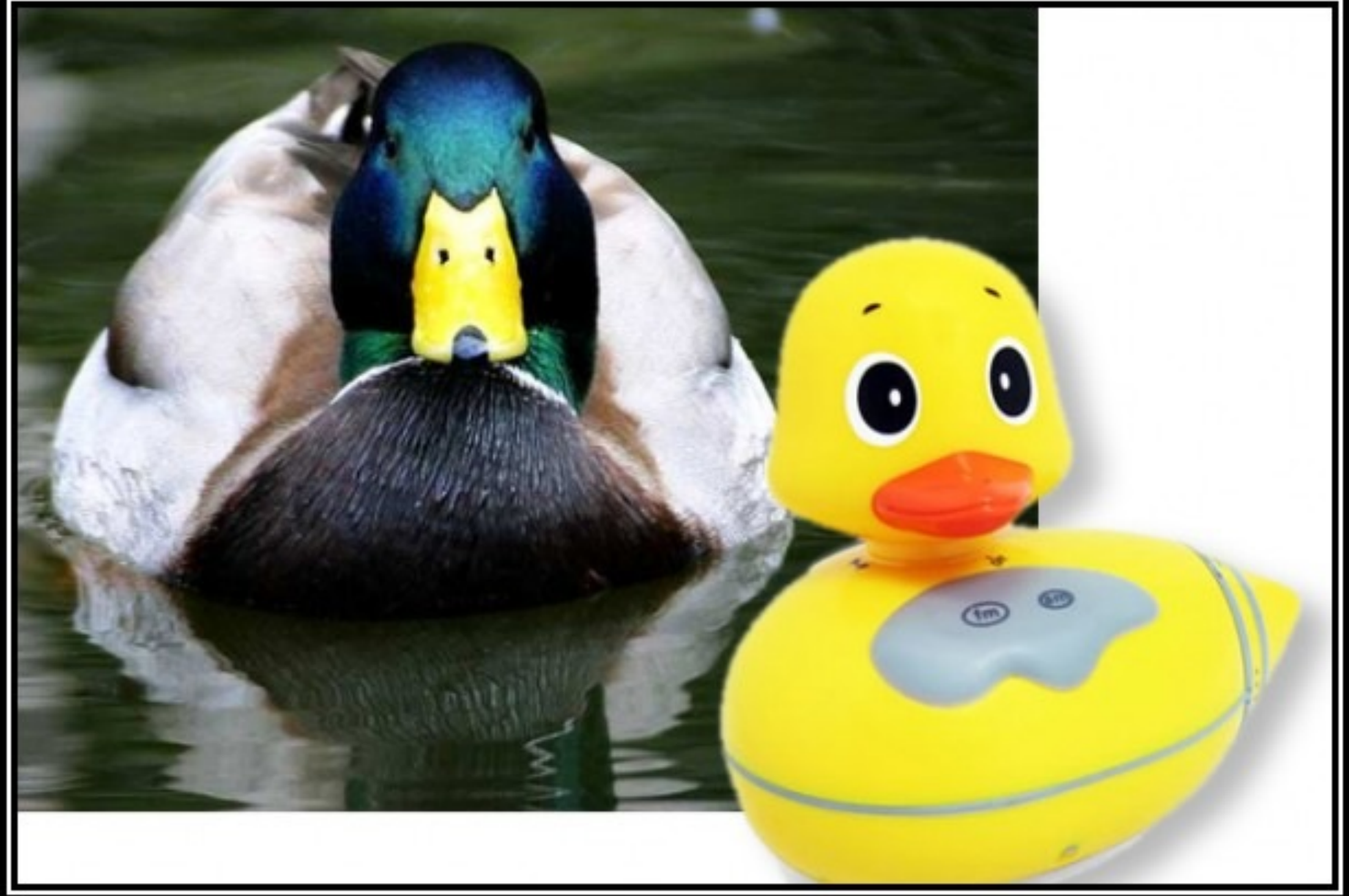
```
#include <Shape.h>

void
DrawAllShapes(Shape* list[],int n)
{
  for(int i=0; i< n; i++)
    list[i]->draw();
}
```


IT University of Copenhagen

S.O.L.I.D.

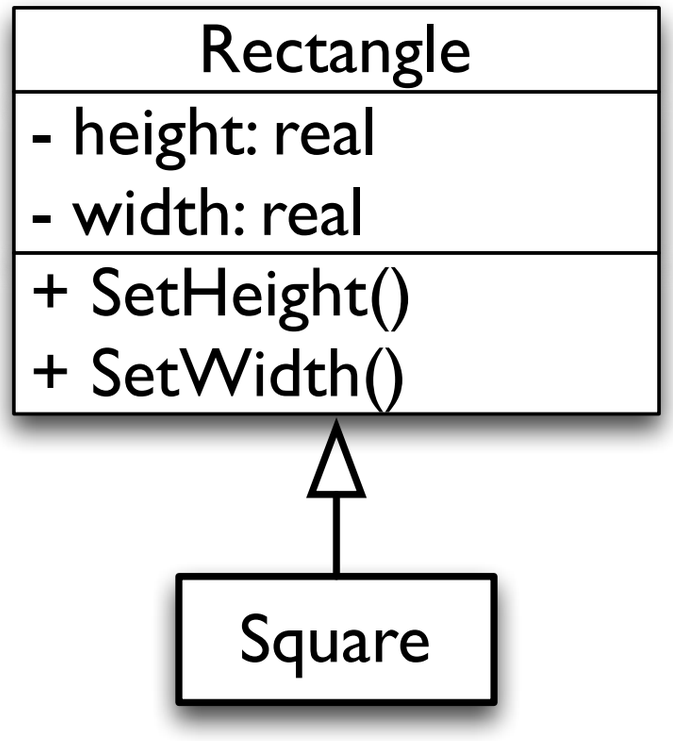
- Single responsibility principle: "a class should only have one, and only one, reason to change".
- Open/closed principle: "software entities should be open for extensions but closed for modifications". (Bertrand Meyer 1988)
- Liskov substitution principle: "derived classes must be usable through the base class interface, without the need for the user to know the difference". (Barbara Liskov 1987)
- Interface segregation principle: "many client-specific interfaces are better than one general-purpose interface".
- Dependency inversion principle: "depend upon abstractions, do not depend upon concretions".



LISKOV SUBSTITUTION PRINCIPLE

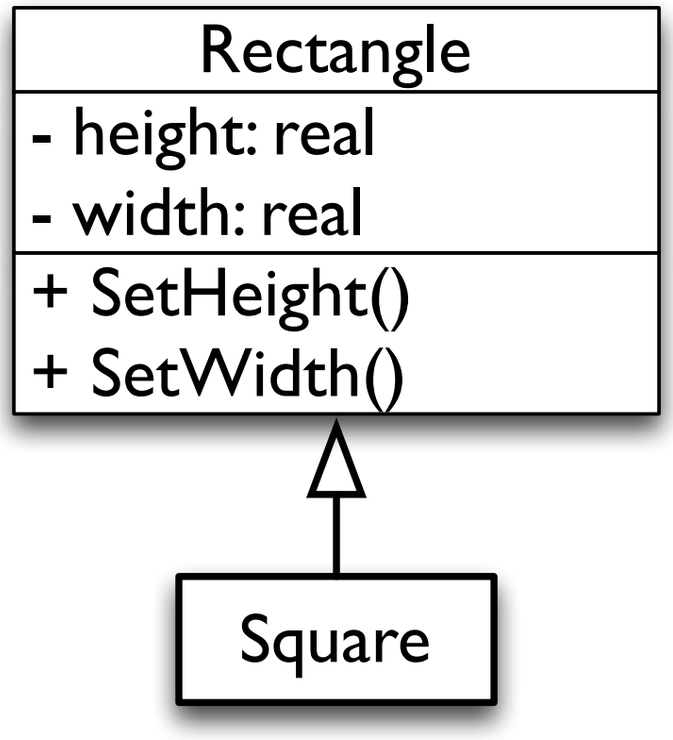
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Liskov substitution principle



- Derived classes should be usable through the base class interface, without the need for the user to know the difference.

Liskov substitution principle



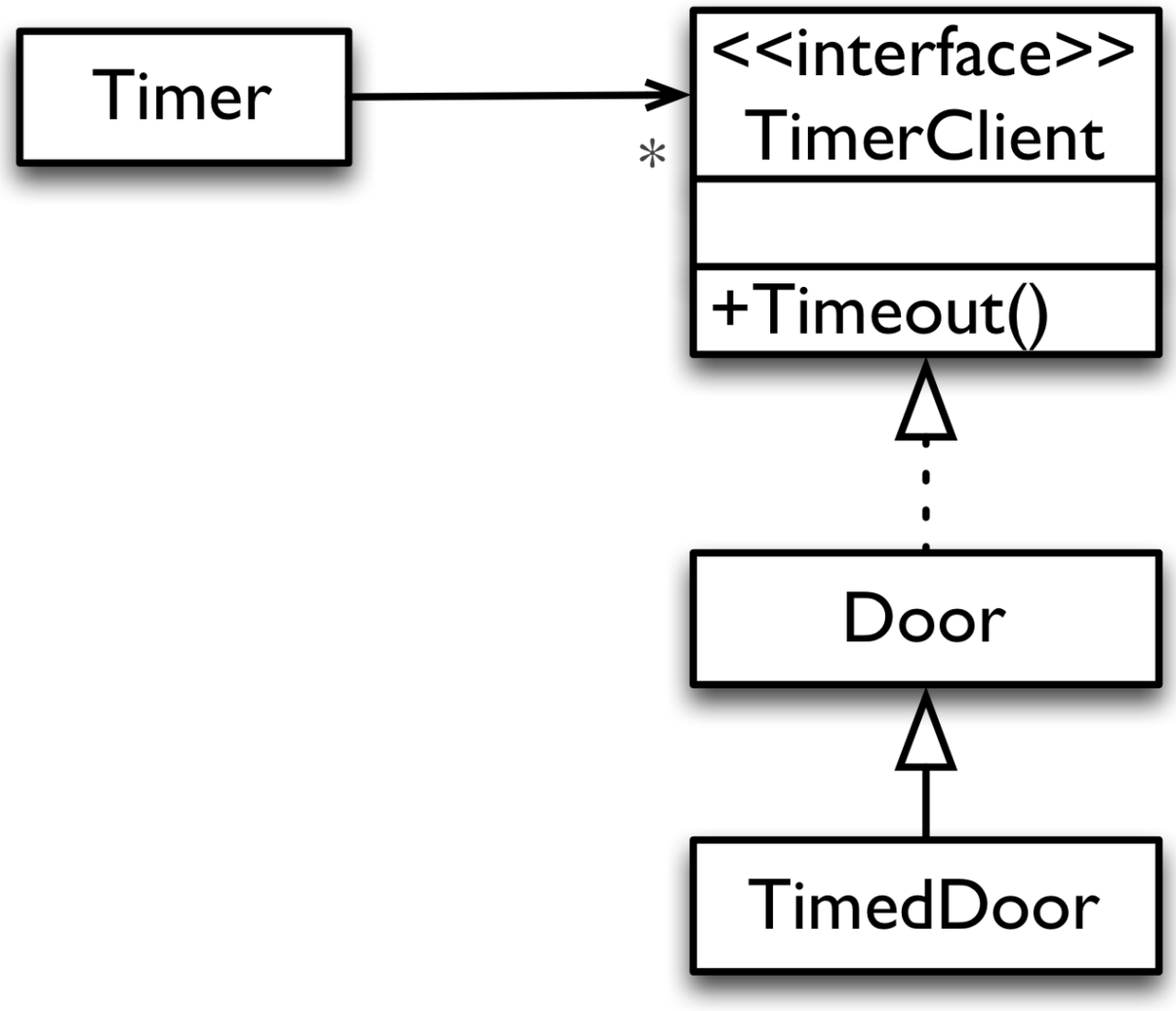
- They do not share the same number of variables, they do not share behaviour, they are not related at least not as siblings and rectangle is not the parent of square.

- The inheritance relation is:
  - the redeclaration of functions and variables in a sub scope

S.O.L.I.D.

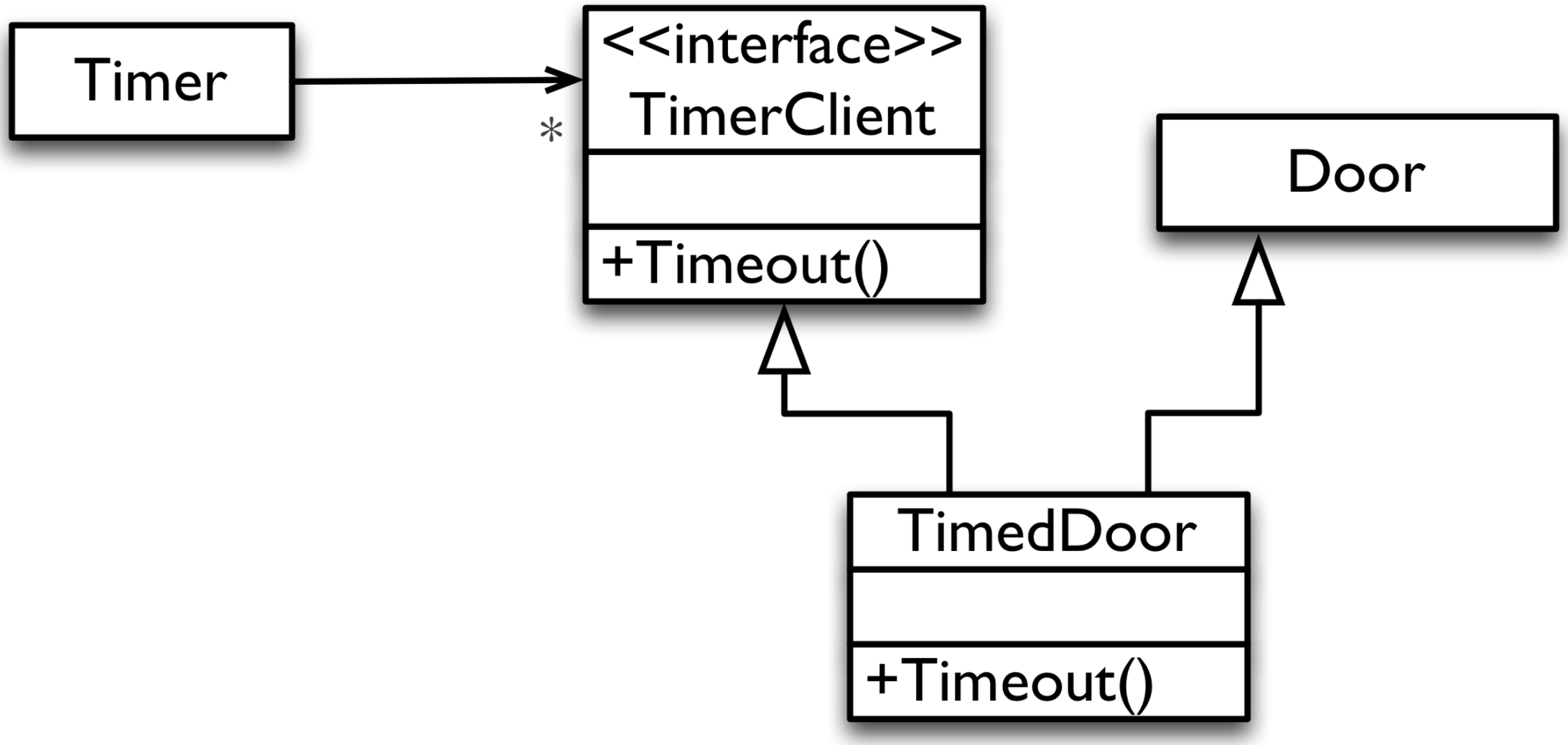
- Single responsibility principle: "a class should only have one, and only one, reason to change".
- Open/closed principle: "software entities should be open for extensions but closed for modifications". (Bertrand Meyer 1988)
- Liskov substitution principle: "derived classes should be usable through the base class interface, without the need for the user to know the difference". (Barbara Liskov 1987)
- Interface segregation principle: "many client-specific interfaces are better than one general-purpose interface".
- Dependency inversion principle: "depend upon abstractions, do not depend upon concretions".

Interface segregation principle

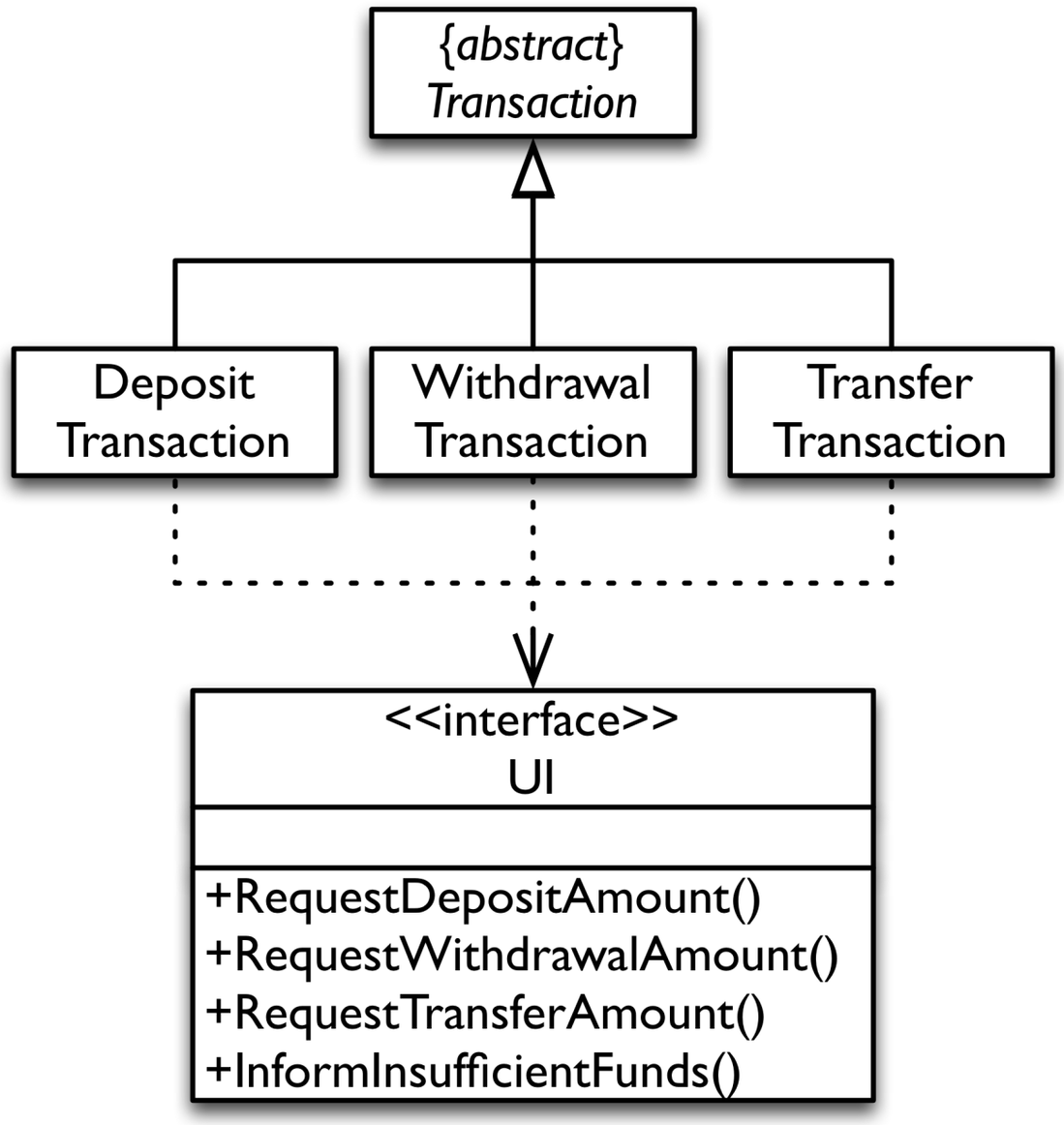


- Many client-specific interfaces are better than one general-purpose interface.

Interface segregation principle

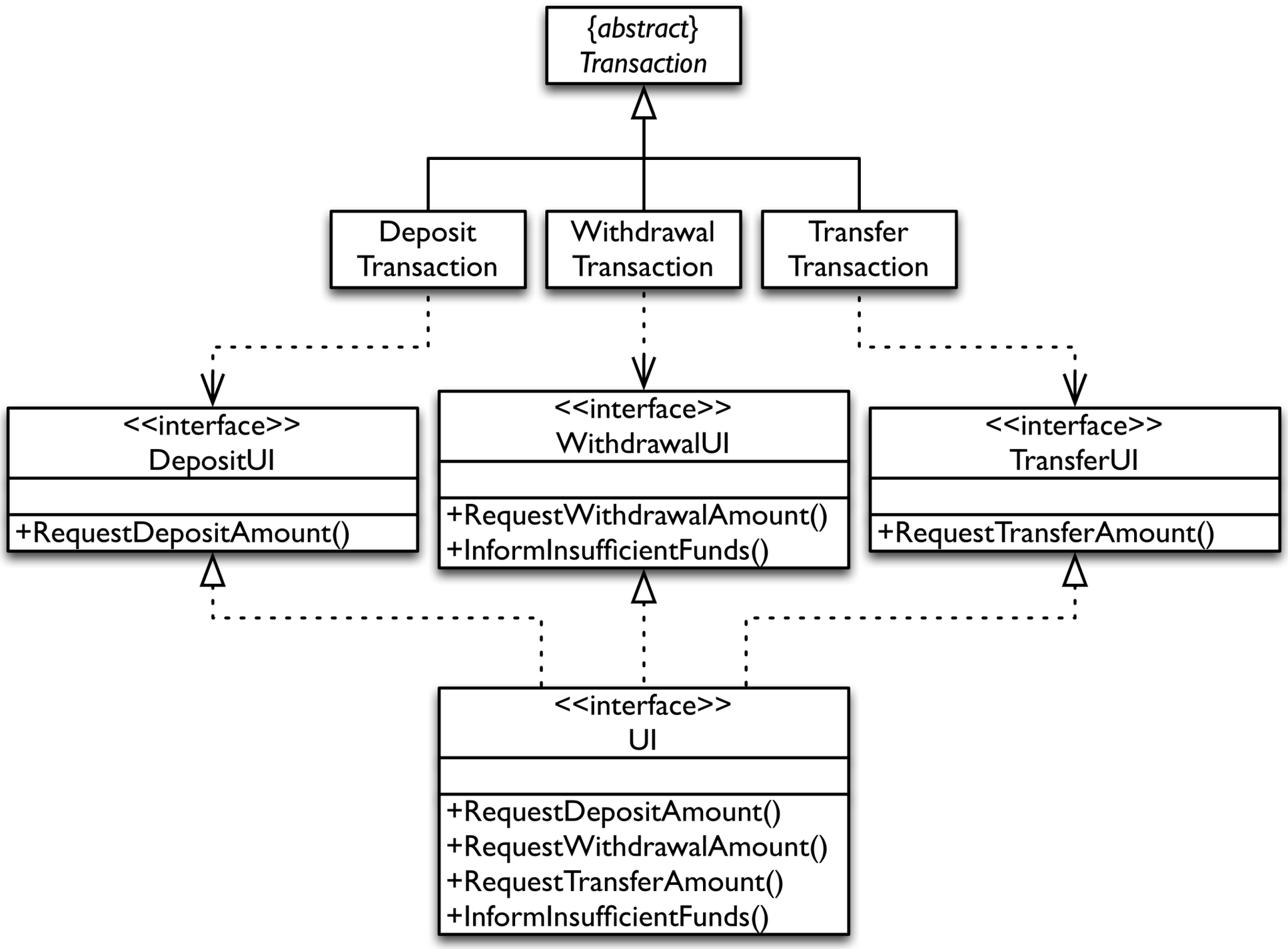


Interface segregation principle



- Many client-specific interfaces are better than one general-purpose interface.

Interface segregation principle



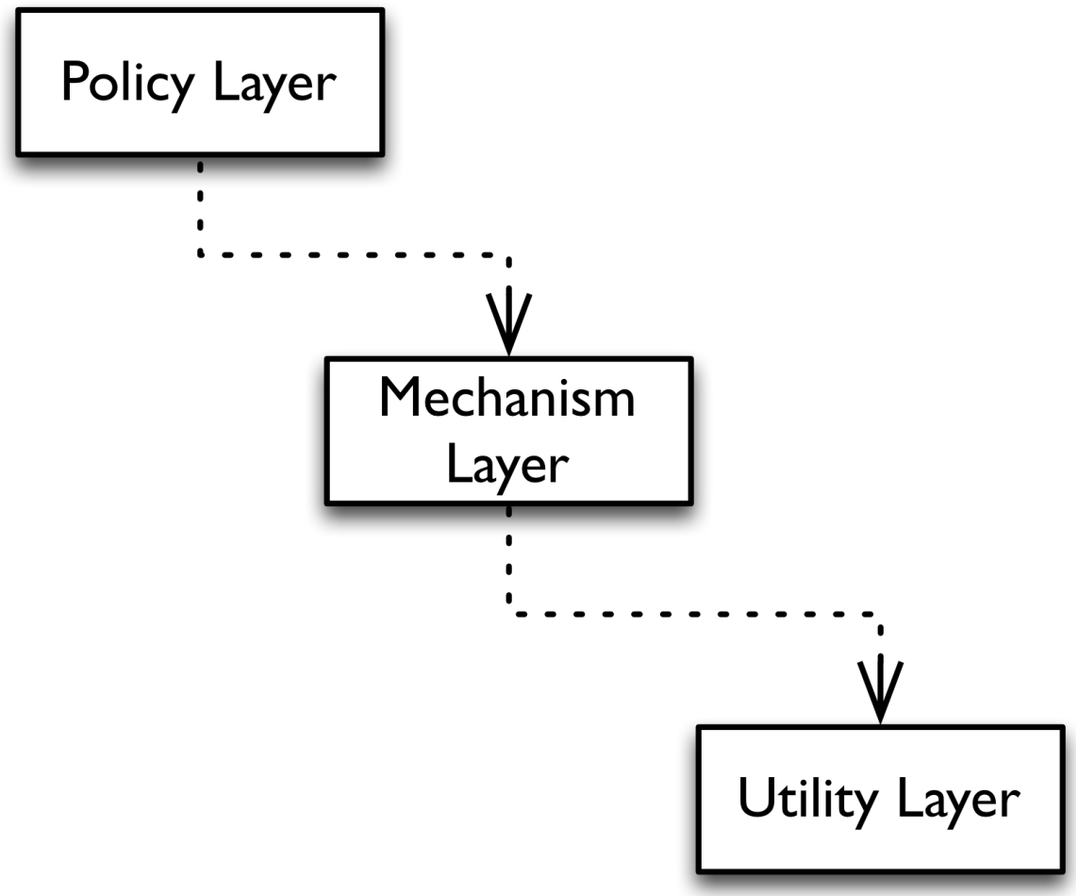
S.O.L.I.D.

- Single responsibility principle: "a class should only have one, and only one, reason to change".
- Open/closed principle: "software entities should be open for extensions but closed for modifications". (Bertrand Meyer 1988)
- Liskov substitution principle: "derived classes should be usable through the base class interface, without the need for the user to know the difference". (Barbara Liskov 1987)
- Interface segregation principle: "many client-specific interfaces are better than one general-purpose interface".
- Dependency inversion principle: "depend upon abstractions, do not depend upon concretions".

DEPENDENCY INVERSION PRINCIPLE

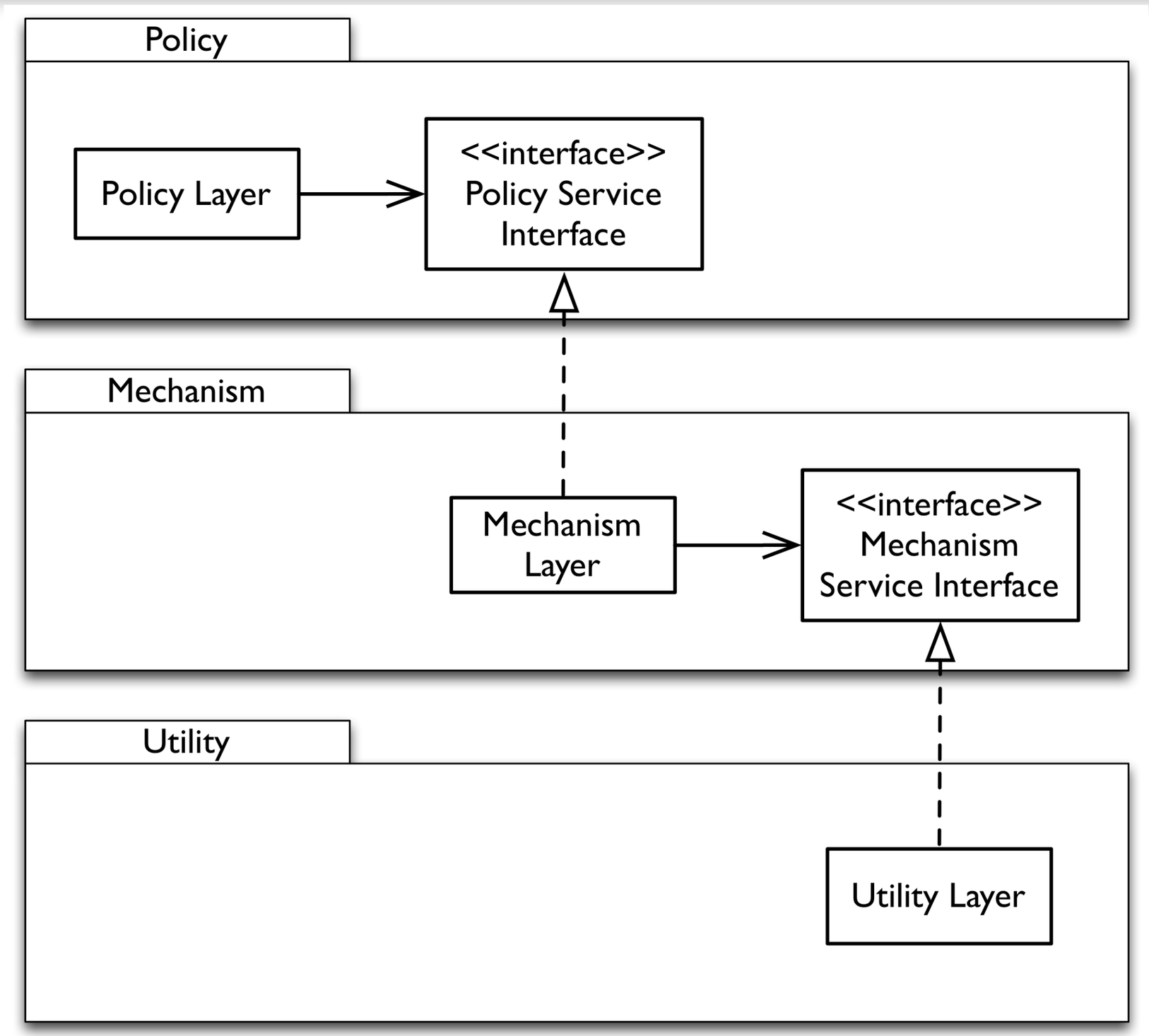
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Dependency inversion principle

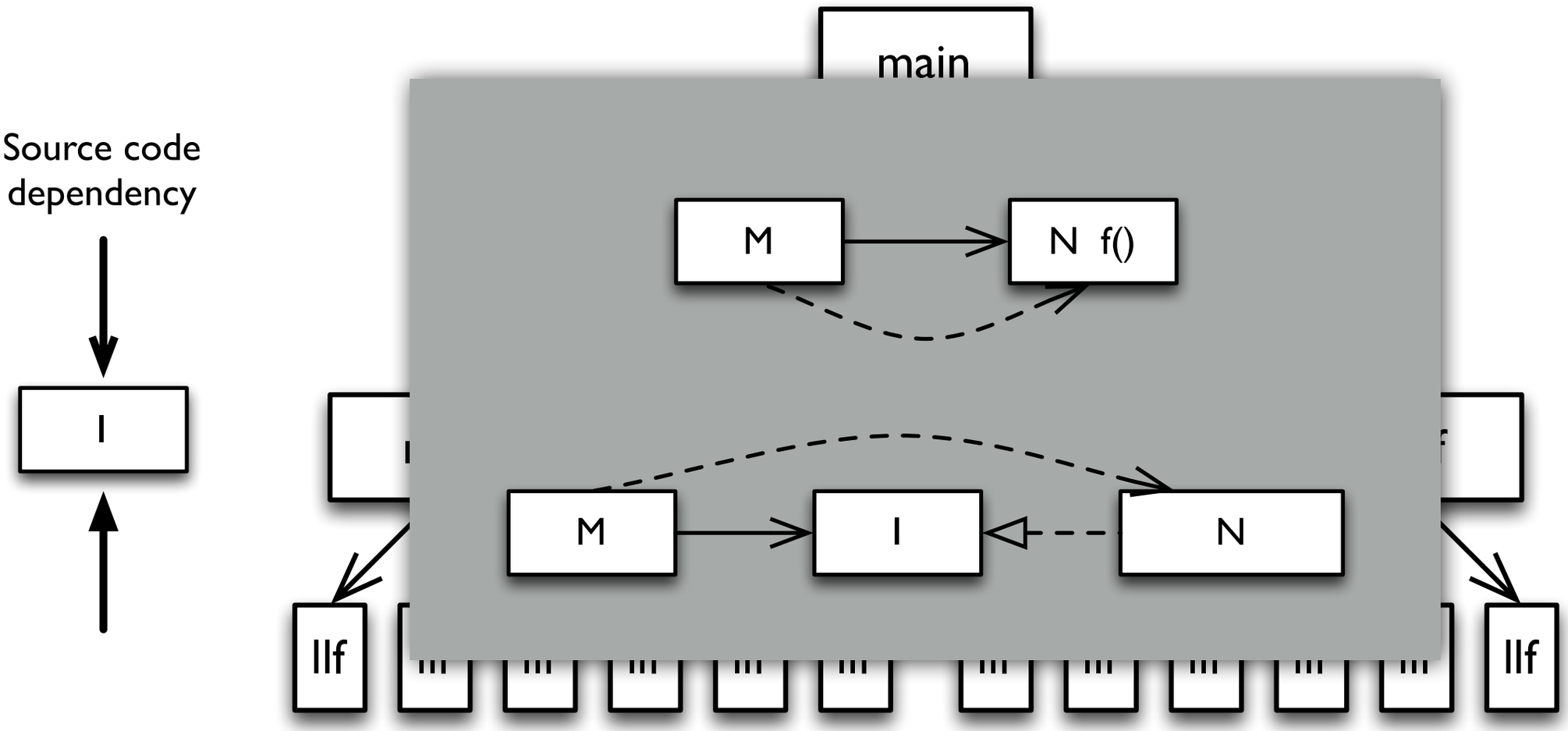


- Depend upon abstractions, do not depend upon concretions.

Dependency inversion principle



!!! polymorphism !!!



Concluding

Key points

- Single responsibility principle: "a class should only have one, and only one, reason to change".
- Open/closed principle: "software entities should be open for extensions but closed for modifications". (Bertrand Meyer 1988)
- Liskov substitution principle: "derived classes should be usable through the base class interface, without the need for the user to know the difference". (Barbara Liskov 1987)
- Interface segregation principle: "many client-specific interfaces are better than one general-purpose interface".
- Dependency inversion principle: "depend upon abstractions, do not depend upon concretions".

Outline

- Literature for the next lectures
  - [OOSE] ch. 6, 7, 8
  - (Optional) [SE9] ch. 6, 7

- Topics covered today:
  - SOLID principles