F. Tip and
M. Weintraub

PRINCIPLES OF SOFTWARE DESIGN

Thanks go to Andreas Zeller for allowing incorporation of his materials

Northeastern University
College of Computer and Information Science
440 Huntington Avenue . 202 West Village H . Boston, MA 02115 . T 617.373.2462 . ccis.northeastern.edu

THE CHALLENGE

1. Software may live much longer than expected

2. Software must be continuously adapted to a changing environment

3. Maintenance takes 50–80% of the cost

Goal: Make software *maintainable* and *reusable* – at little or no cost

USE THE PRINCIPLES OF OBJECT-ORIENTED DESIGN TO ACHIEVE THE GOAL

1. Abstraction
2. Encapsulation
3. Modularity
4. Hierarchy

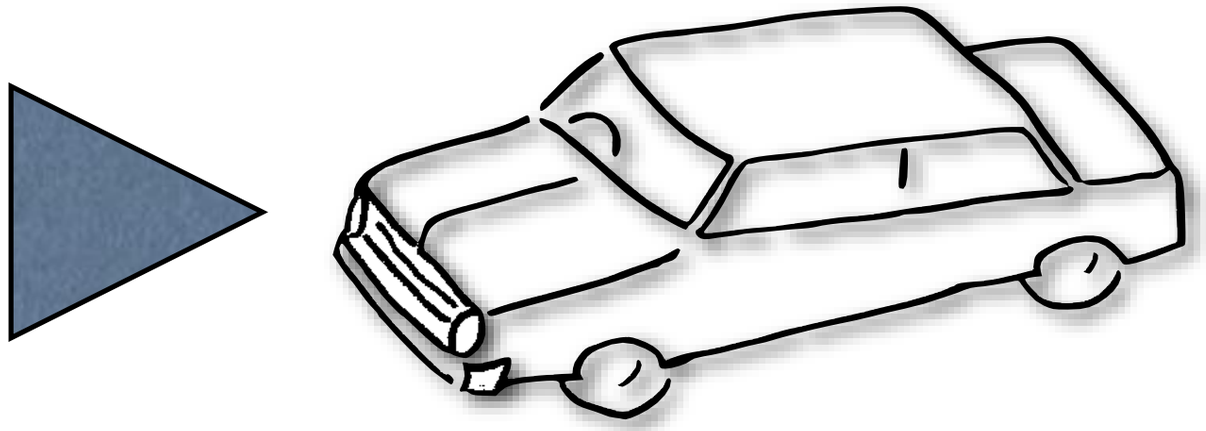
Goal: *Maintainability* and *Reusability*

PRINCIPLES OF OBJECT-ORIENTED DESIGN

1. Abstraction
2. Encapsulation
3. Modularity
4. Hierarchy

ABSTRACTION

Concrete Object



General Principle

ABSTRACTION...

1. Highlights *common properties* of objects

2. Distinguishes *important* and *unimportant* properties

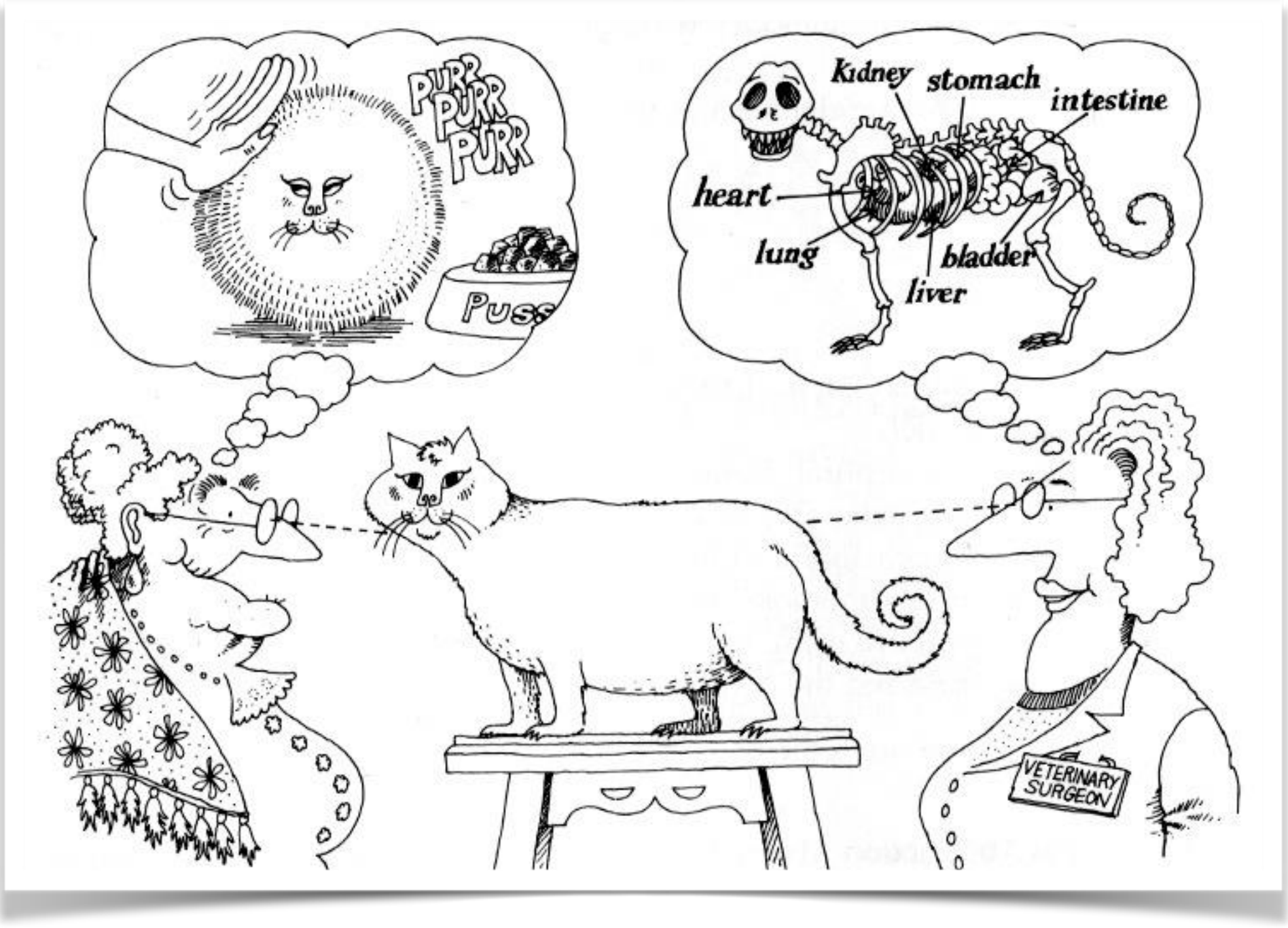
3. Must be understood even without a concrete object

ABSTRACTION

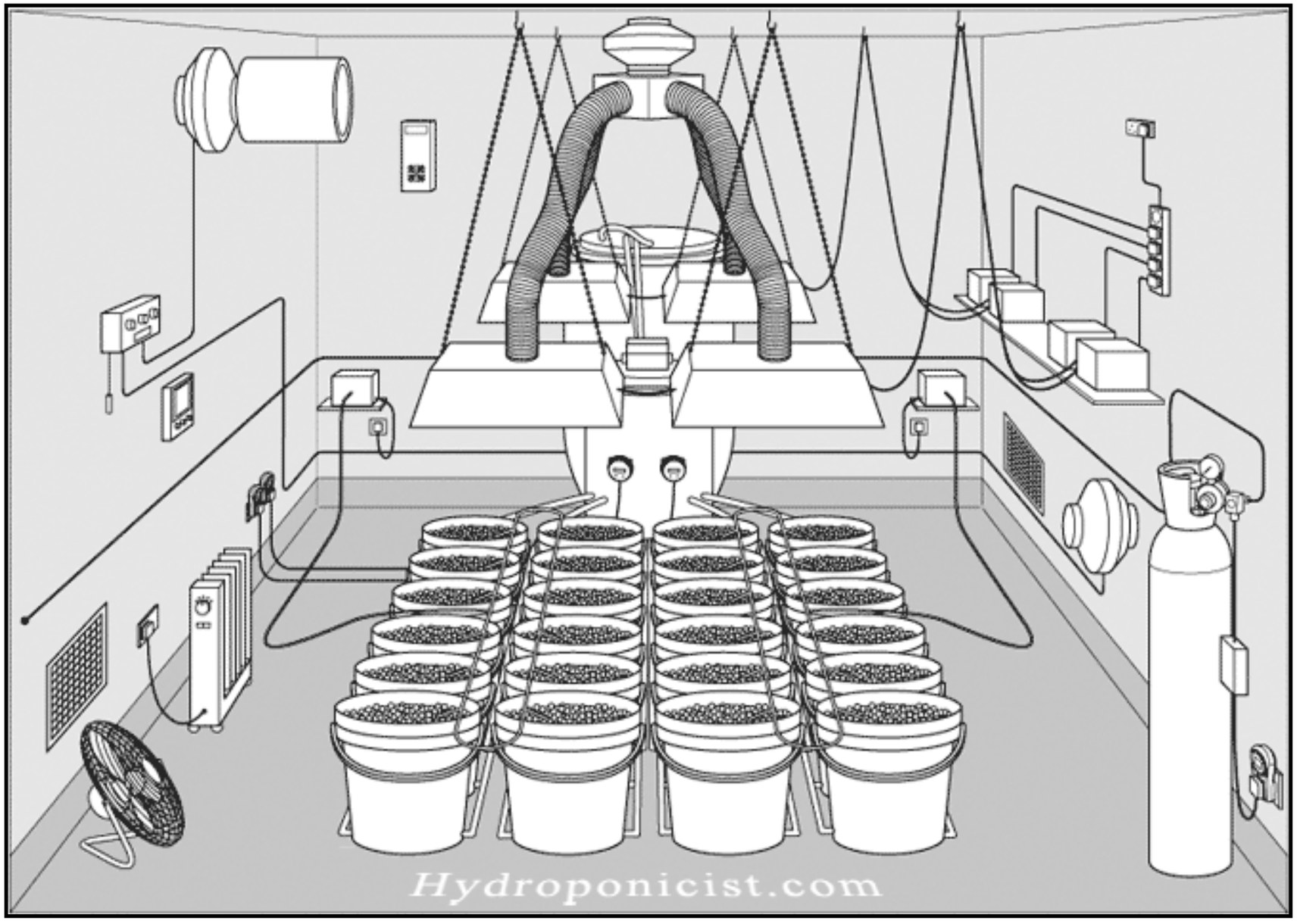
“An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer”

From “Object Oriented Design with Applications” by Grady Booch

PERSPECTIVES



EXAMPLE: SENSORS



AN ENGINEER'S SOLUTION

```
void check_temperature() {
    // see specs AEG sensor type 700, pp. 53

    short *sensor = 0x80004000;
    short *low    = sensor[0x20];
    short *high   = sensor[0x21];
    int temp_celsius = low + high * 256;

    if (temp_celsius > 50) {
        turn_heating_off()
    }
}
```

C code where values read by a sensor are directly mapped to memory locations

ABSTRACT SOLUTION

```
interface Temperature { … }
interface Location { … }
```

All implementation details are *hidden*

```
class TemperatureSensor {
    public TemperatureSensor(Location){ … }

    public void calibrate(Temperature actual){ … }
    public Temperature currentTemperature(){ … }
    public Location location(){ … }

    // private methods below
}
```

MORE ABSTRACTION

IT'S A PROJECTION OF A SLIDE OF A PHOTO OF A PAINTING OF A PIPE

PRINCIPLES OF OBJECT-ORIENTED DESIGN

1. Abstraction – hide details
2. Encapsulation
3. Modularity
4. Hierarchy

PRINCIPLES OF OBJECT-ORIENTED DESIGN

1. Abstraction – Hide details
2. Encapsulation
3. Modularity
4. Hierarchy

ENCAPSULATION

- No part of a complex system should depend on internal details of another

Goal: keep software changes *local*

Information hiding: Internal details (state, structure, behavior) become the object's *secret*

GRADY BOOCH ON ENCAPSULATION

"Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and its behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation."

Grady Booch, Object-Oriented Analysis and Design with Applications, Addison-Wesley, 2007, p. 51-52

AN ACTIVE SENSOR

notified when temperature changes

```
class ActiveSensor {
  public ActiveSensor(Location)

  public void calibrate(Temperature actual){ … }
  public Temperature currentTemperature(){ … }
  public Location location(){ … }

  public void register(ActiveSensorObserver o){ … }

    // private methods below…
}
```

Callback management is the sensor's secret and this illustrates how the "Observer" design pattern is used to avoid giving external parties access to internal state of the ActiveSensor

ANTICIPATING CHANGE

Features you expect will change should be *isolated* in specific components

- Number literals
- String literals
- Presentation and interaction

NUMBER LITERALS

```
int a[100]; for (int i = 0; i <= 99; i++) a[i] = 0;
```

NUMBER LITERALS

```
int a[100]; for (int i = 0; i <= 99; i++) a[i] = 0;
```

```
int SIZE = 100;
int a[SIZE]; for (int i = 0; i < SIZE; i++) a[i] = 0;
```

~~int ONE_HUNDRED = 100;~~
~~int a[ONE_HUNDRED]; ...~~

NUMBER LITERALS

```
double sales_price = net_price * 1.06;
```

NUMBER LITERALS

```
double sales_price = net_price * 1.06;
```

```
final double SALES_TAX = 1.06;
double sales_price = net_price * SALES_TAX;
```

STRING LITERALS

```
if (sensor.temperature() > 100)
    System.out.println(“Water is boiling!”);
```

STRING LITERALS

```
if (sensor.temperature() > 100)
    System.out.println(“Water is boiling!”);
```

```
if (sensor.temperature() > BOILING_POINT)
    System.out.println(message(BOILING_WARNING,
                               “Water is boiling!”);

if (sensor.temperature() > BOILING_POINT)
    alarm.handle_boiling();
```

PRINCIPLES OF OBJECT-ORIENTED DESIGN

1. Abstraction – Hide details
2. Encapsulation – Keep changes local
3. Modularity
4. Hierarchy

PRINCIPLES OF OBJECT-ORIENTED DESIGN

1. Abstraction – Hide details
2. Encapsulation – Keep changes local
3. Modularity
4. Hierarchy

MODULARITY

Basic idea: Partition a system such that parts can be designed and revised independently (“divide and conquer”)

System is partitioned into *modules,* with each one fulfilling a specific task

Modules should be changeable and reuseable independent of other modules

GRADY BOOCH ON MODULARITY

“Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.”

MODULE BALANCE

Goal 1: Modules should *hide information* – and expose as little as possible

Goal 2: Modules should *cooperate* – and therefore must exchange information

These goals conflict with each other

PRINCIPLES OF MODULARITY

High cohesion	Modules should contain functions that logically belong together
Weak coupling	Changes to modules should not affect other modules
Law of Demeter	Talk only to friends

HIGH COHESION

1. Modules should contain functions that logically belong together

2. Achieved by grouping functions that work on the same data

3. “Natural” grouping in object oriented design

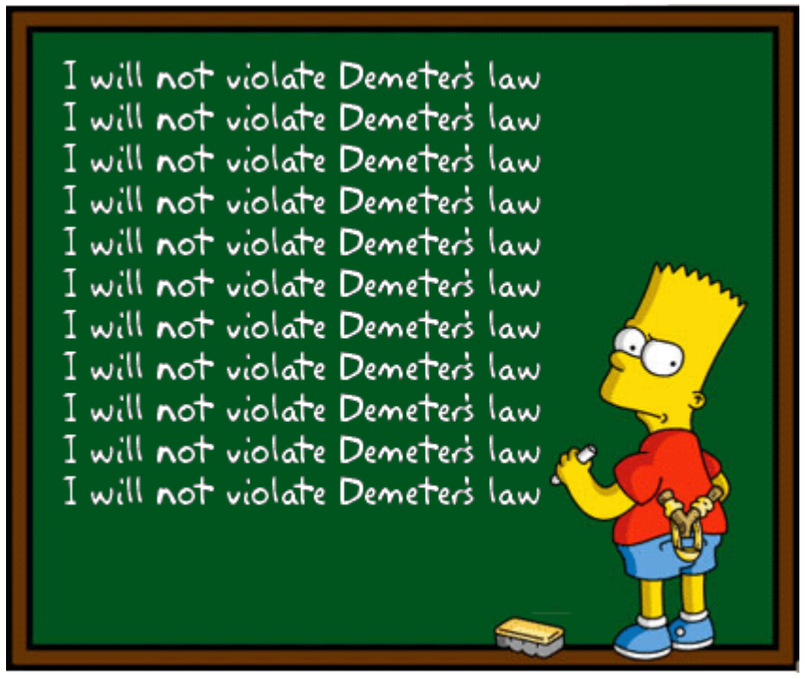
WEAK COUPLING

Changes in modules should not impact other modules

Achieved via

1. Information hiding
2. Depending on as few modules as possible

LAW OF DEMETER
(OR: PRINCIPLE OF LEAST KNOWLEDGE)



Basic idea: Assume as little as possible about other modules

Approach: Restrict method calls to *friends*

Demeter (aka Ceres) is the Greek mythical goddess of the harvest, and she presided also over the sacred law and the cycle of life and death.

Proposed by Holland, Lieberherr, and Riel at Northeastern University in 1988

LoD: CALL YOUR FRIENDS

A method `M` of an object `O` should only call methods of

1. `O` itself
2. `M`'s parameters
3. any objects created in `M`
4. `O`'s direct component objects

"single dot rule"

DEMETER: EXAMPLE

```
class Uni {
    Prof boring = new Prof();
    public Prof getProf() { return boring; }
    public Prof getNewProf() { return new Prof(); }
}

class Test {
    Uni uds = new Uni();
    public void one() { uds.getProf().fired(); }
    public void two() { uds.getNewProf().hired(); }
}
```

DEMETER: EXAMPLE

```
class Uni {
    Prof boring = new Prof();
    public Prof getProf() { return boring; }
    public Prof getNewProf() { return new Prof(); }
    public void fireProf(...) { ... }
}


class BetterTest {
    Uni uds = new Uni();
    public void betterOne() { uds.fireProf(...); }

}
```

DEMETER EFFECTS

1. Reduces coupling between modules

2. Disallow direct access to parts

3. Limit the number of accessible classes

4. Reduce dependencies

5. Results in several new wrapper methods
   - ✦ *“Demeter transmogrifiers”*

PRINCIPLES OF OBJECT-ORIENTED DESIGN

1. Abstraction – Hide details

2. Encapsulation – Keep changes local

3. Modularity – Control information flow
   - ✦ high cohesion
   - ✦ weak coupling
   - ✦ talk only to friends

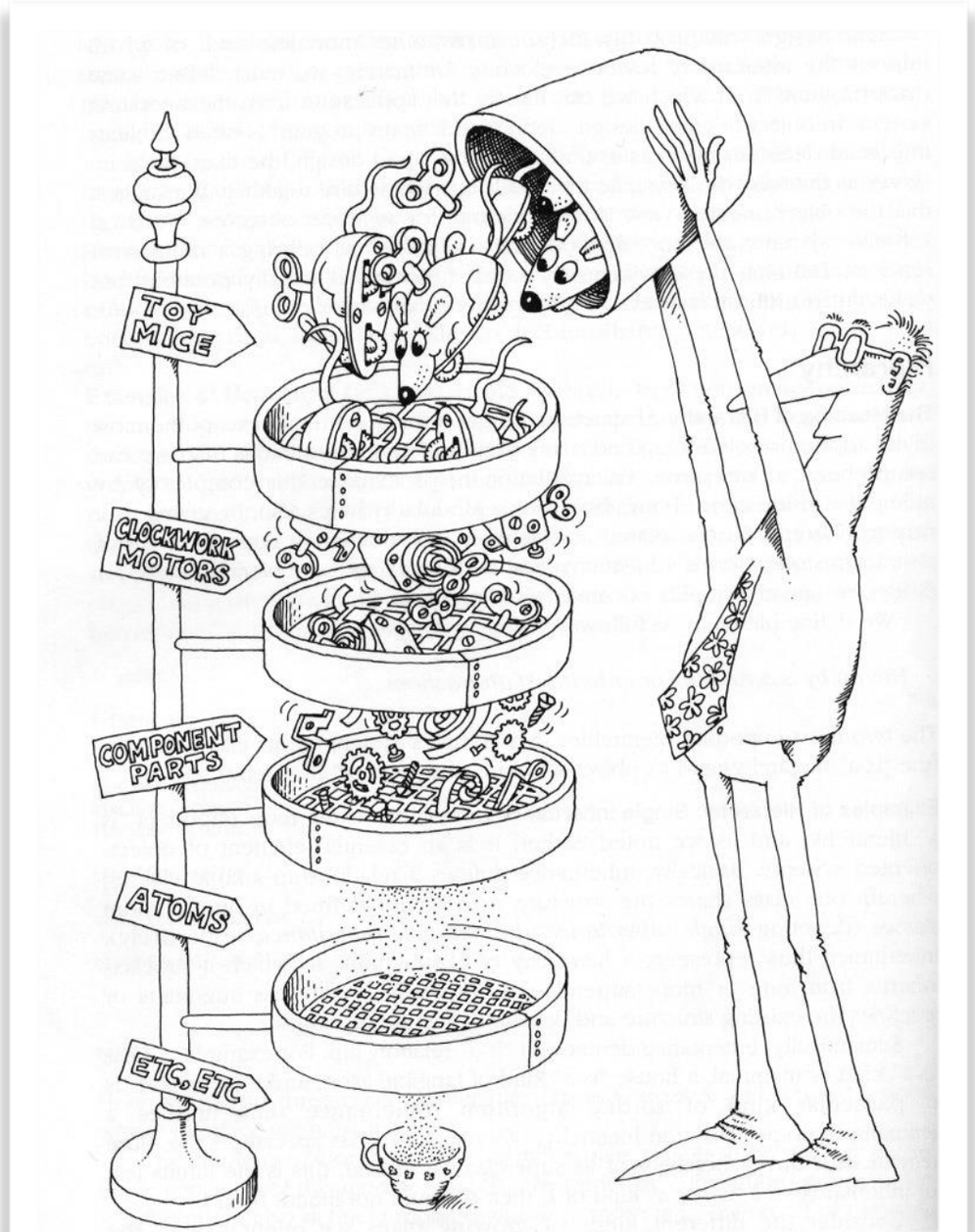
4. Hierarchy

PRINCIPLES OF OBJECT-ORIENTED DESIGN

1. Abstraction – Hide details
2. Encapsulation – Keep changes local
3. Modularity – Control information flow
   - ✦ High cohesion
   - ✦ weak coupling
   - ✦ talk only to friends
4. Hierarchy

HIERARCHY

“Hierarchy is a ranking or ordering of abstractions.”



CENTRAL HIERARCHIES

1. “has-a” hierarchy – *Aggregation* of abstractions
   - ✦A *car* has three to four *wheels*

1. “is-a” hierarchy – *Generalization* across abstractions
   - ✦An *ActiveSensor* is a *TemperatureSensor*

HIERARCHY PRINCIPLES

Open/Close Principle

Classes should be open for extensions

Liskov Substitution Principle

Subclasses should not require more, and not deliver less

Dependency Principle

Classes should only depend on abstractions

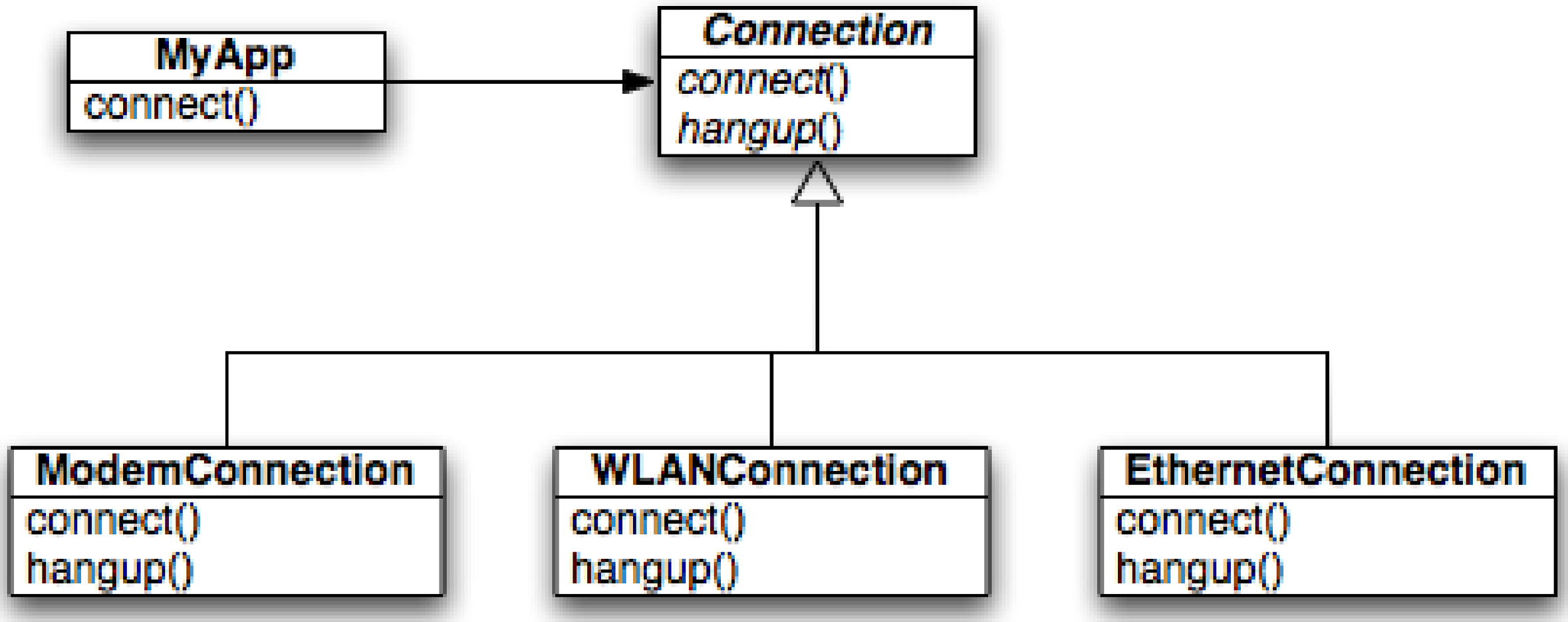
OPEN/CLOSE PRINCIPLE

- A class should be *open* for extension, but *closed* for changes
- Achieved via *inheritance* and *dynamic binding*

AN INTERNET CONNECTION

```
void connect() {
    if (connection_type == MODEM_56K)
    {
        Modem modem = new Modem();
        modem.connect();
    }
    else if (connection_type == ETHERNET) …
    else if (connection_type == WLAN) …
    else if (connection_type == UMTS) …
}
```

SOLUTION WITH HIERARCHIES



AN INTERNET CONNECTION

```
abstract class Connection {
   abstract int connect();
   abstract int hangup();
}
```

```
class EthernetConnection extends Connection {
 int connect() {// does Ethernet connection; }
}

class ModemConnection extends Connection {
 int connect() {// does dial-up connection; }
}
…
```

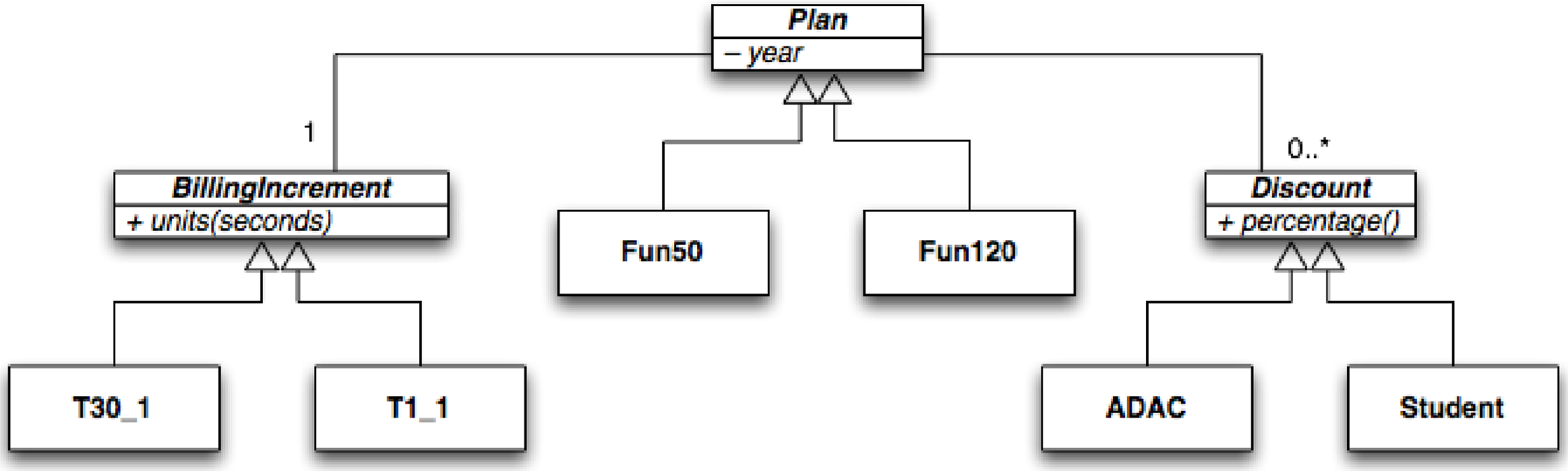
CONSIDER BILLING PLANS

```
enum { FUN50, FUN120, FUN240, ... } plan;
enum { STUDENT, ADAC, ADAC_AND_STUDENT ... } special;
enum { PRIVATE, BUSINESS, ... } customer_type;
enum { T60_1, T60_60, T30_1, ... } billing_increment;

int compute_bill(int seconds)
{
    if (customer_type == BUSINESS)
        billing_increment = T1_1;
    else if (plan == FUN50 || plan == FUN120)
        billing_increment = T60_1;
    else if (plan == FUN240 && contract_year < 2011)
        billing_increment = T30_1;
    else
        billing_increment = T60_60;

    if (contract_year >= 2011 && special != ADAC)
        billing_increment = T60_60;
    // etc.etc.
```

HIERARCHY SOLUTION



You can add a new plan at any time!

HIERARCHY PRINCIPLES

- Open/Close principle – Classes should be open for extensions
- Liskov substitution principle – Subclasses should not require more, and not deliver less
- Dependency principle – Classes should only depend on abstractions

LISKOV SUBSTITUTION PRINCIPLE

An object of a superclass should always be substitutable by an object of a subclass:

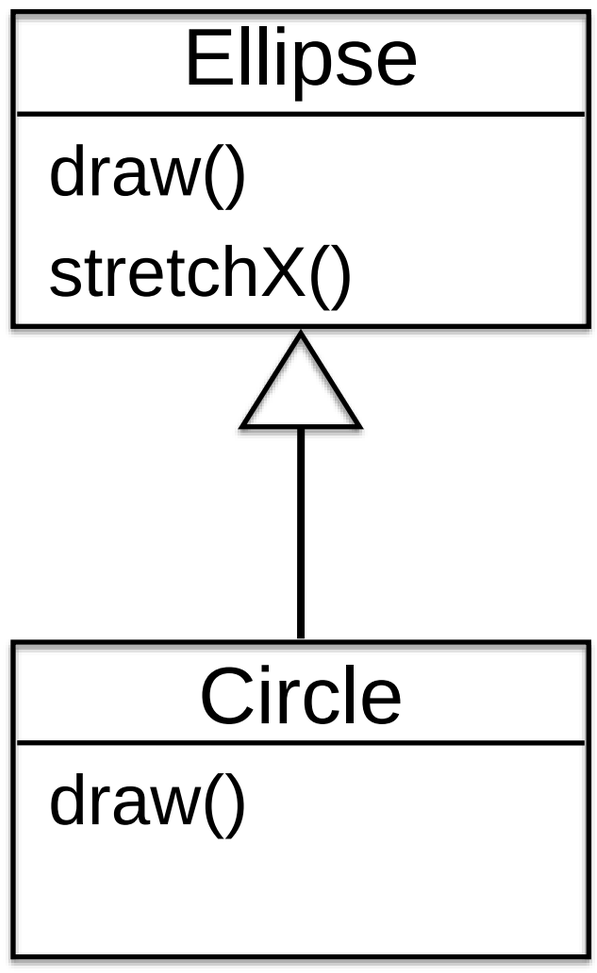
- ✦Same or weaker preconditions
- ✦Same or stronger postconditions

Derived methods should *not assume more or deliver less*

CIRCLE VS ELLIPSE

- Every circle is an ellipse
- Does this hierarchy make sense?

  No, as a circle requires more and delivers less



HIERARCHY PRINCIPLES

- Open/Close principle – Classes should be open for extensions
- Liskov substitution principle – Subclasses should not require more, and not deliver less
- Dependency principle – Classes should only depend on abstractions

DEPENDENCY PRINCIPLE

A class should only depend on *abstractions* – never on concrete subclasses *(dependency inversion principle)*

This principle can be used to *break* dependencies

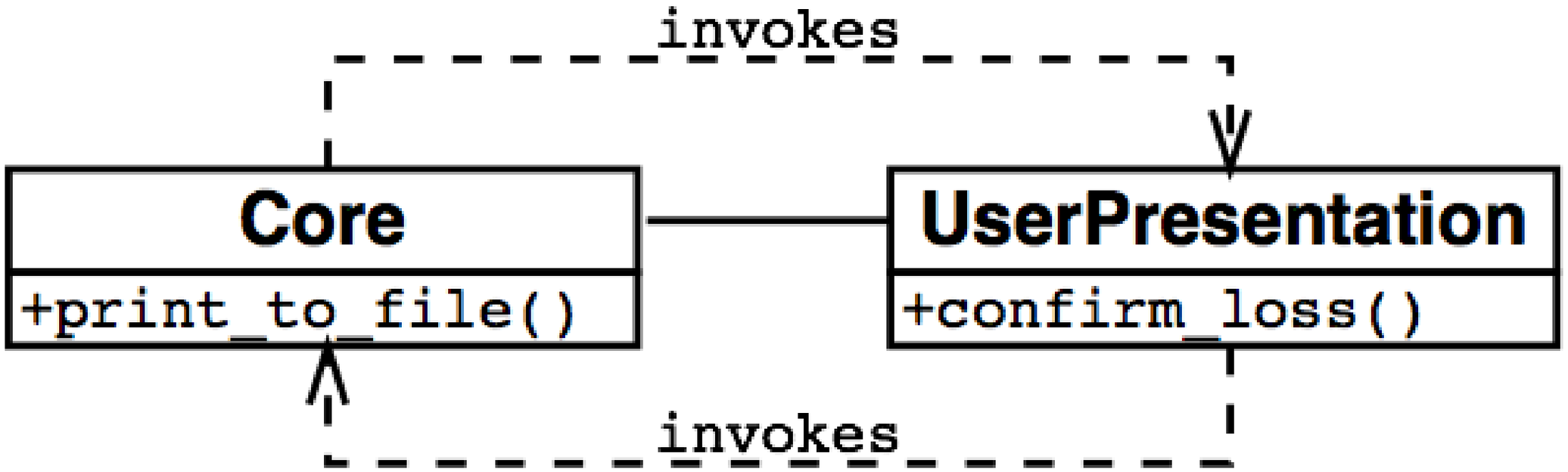
DEPENDENCY

```
// Print current Web page to FILENAME after user clicks “print."

void print_to_file(string filename)
{
    if (path_exists(filename))
    {
        // FILENAME exists;
        // ask user to confirm overwrite in UserPresentation
        bool confirmed = confirm_loss(filename);
        if (!confirmed)
            return;
    }

    // Proceed printing to FILENAME
    ...
}
```

CYCLIC DEPENDENCY



constructing, testing, reusing individual modules becomes impossible!

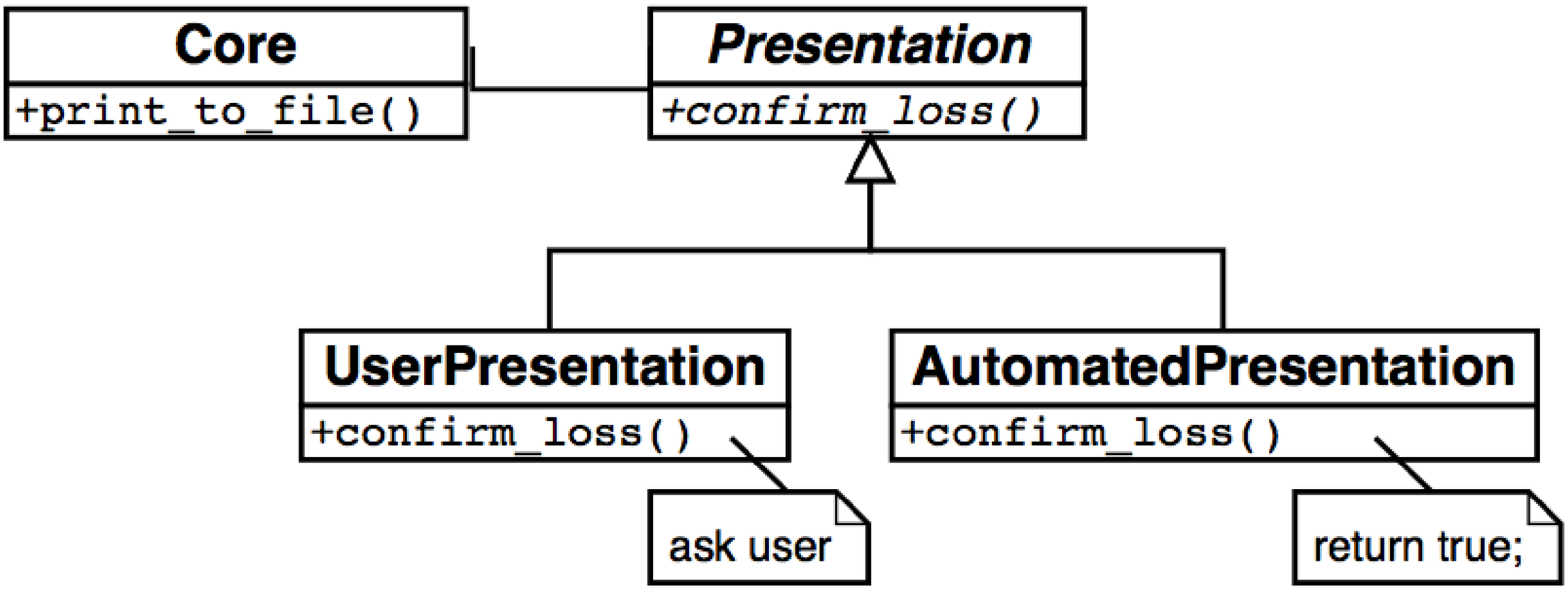
DEPENDENCY

```
// Print current Web page to FILENAME after user clicks “print."

void print_to_file(string filename, Presentation p)
{
    if (path_exists(filename))
    {
        // FILENAME exists;
        // ask user to confirm overwrite
        bool confirmed = p.confirm_loss(filename);
        if (!confirmed)
            return;
    }

    // Proceed printing to FILENAME
    ...
}
```

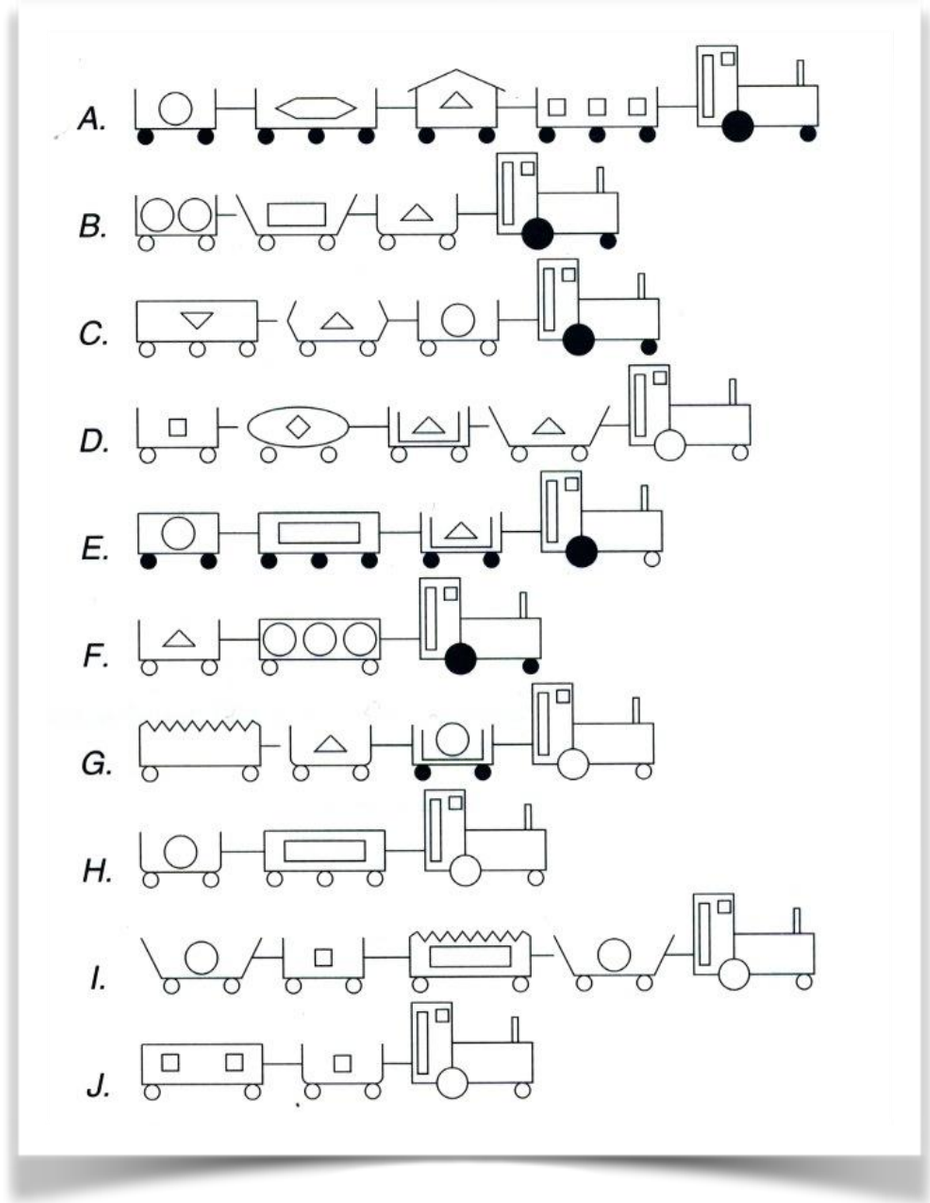
DEPENDING ON ABSTRACTION



CHOOSING ABSTRACTION

1. Which is the “dominant” abstraction?

2. How does this choice impact the remaining system?



PRINCIPLES OF OBJECT-ORIENTED DESIGN

- Abstraction – Hide details
- Encapsulation – Keep changes local
- Modularity – Control information flow
  - high cohesion
  - weak coupling
  - talk only to friends
- Hierarchy – Order abstractions
  - classes open for extensions, closed for changes
  - subclasses that do not require more or deliver less
  - depend only on abstractions

PRINCIPLES OF OBJECT-ORIENTED DESIGN

- Abstraction – Hide details
- Encapsulation – Keep changes local
- Modularity – Control information flow
  - Goal: Maintainability and Reusability
  - weak coupling
  - talk only to friends
- Hierarchy – Order abstractions
  - classes open for extensions, closed for changes
  - subclasses that do not require more or deliver less
  - depend only on abstractions