



Maintainable Software

Software Engineering
Andreas Zeller, Saarland University

The Challenge

- Software may live much longer than expected
- Software must be continuously adapted to a changing environment
- Maintenance takes 50–80% of the cost
- Goal: Make software *maintainable* and *reusable* – at little or no cost

Software Maintenance

- **Corrective maintenance:** Fix bugs
- **Adaptive maintenance:** adapt system to the environment it operates
- **Perfective maintenance:** adapt to new or changed requirements
- **Preventive maintenance:** increase quality or prevent future bugs from

Software Maintenance

- **Follow principles of object-oriented design**
- **Follow guidelines for maintainable software**

Principles

of object-oriented design

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

Goal: *Maintainability* and *Reusability*

Principles

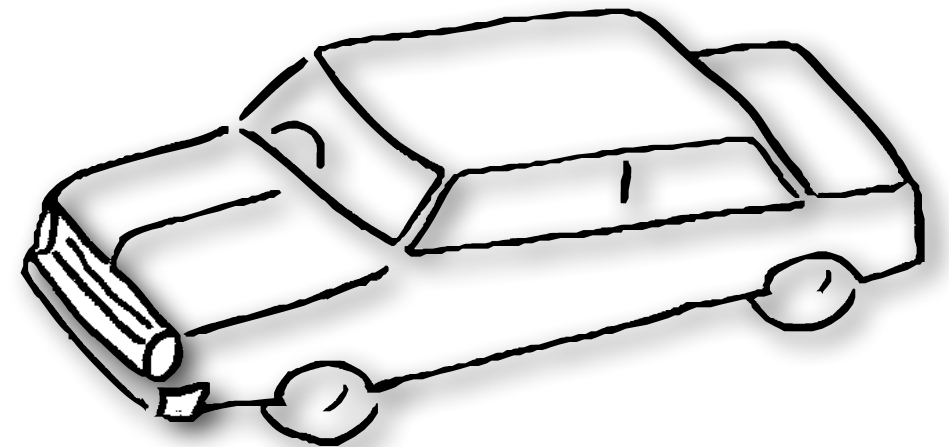
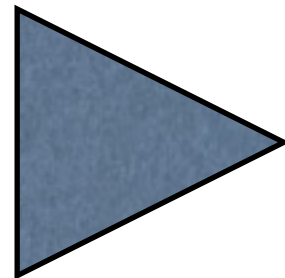
of object-oriented design

- **Abstraction**
- Encapsulation
- Modularity
- Hierarchy

Abstraction



Concrete Object



General Principle

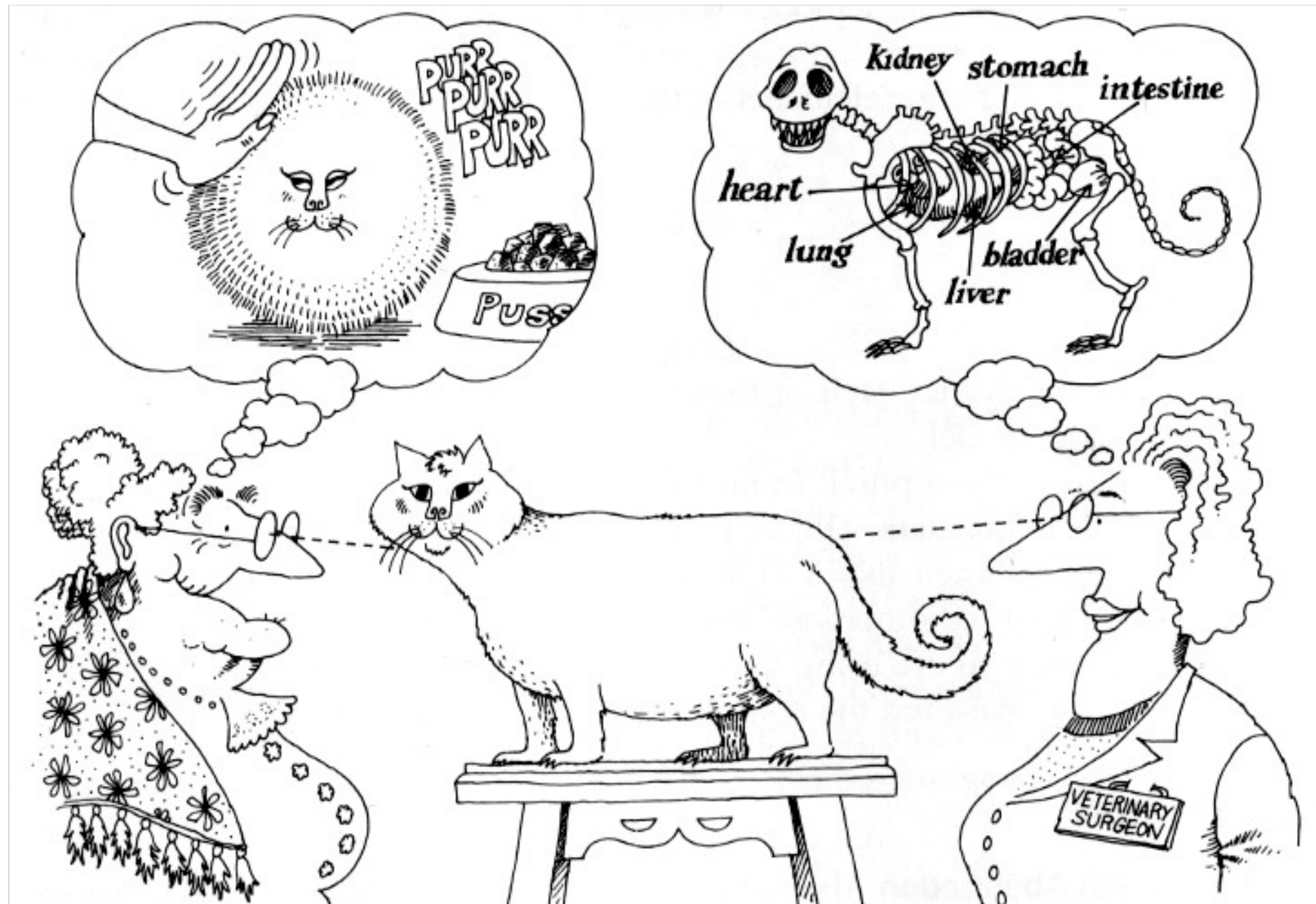
Abstraction...

- Highlights *common properties* of objects
- Distinguishes *important* and *unimportant* properties
- Must be understood even without a concrete object

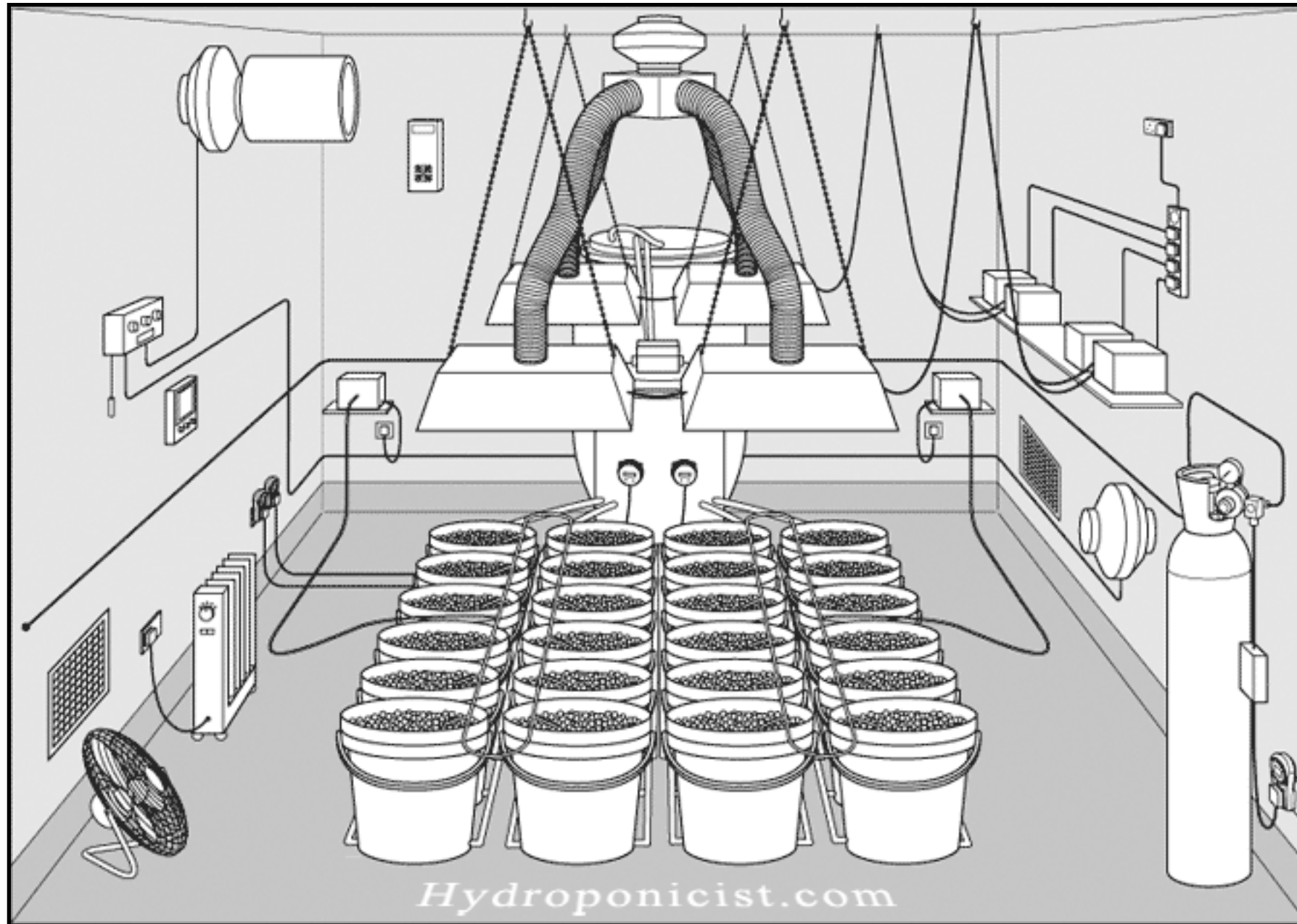
Abstraction

“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”

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()  
    }  
}
```

Abstract Solution

```
typedef float Temperature;  
typedef int Location;
```

```
class TemperatureSensor {  
public:  
    TemperatureSensor(Location);  
    ~TemperatureSensor();  
  
    void calibrate(Temperature actual);  
    Temperature currentTemperature() const;  
    Location location() const;  
  
private: ...  
}
```

All implementation
details are *hidden*

More Abstraction



Ceci n'est pas une pipe.

Principles

of object-oriented design

- **Abstraction – hide details**
- Encapsulation
- Modularity
- Hierarchy

Principles

of object-oriented design

- Abstraction – Hide details
- **Encapsulation**
- Modularity
- Hierarchy

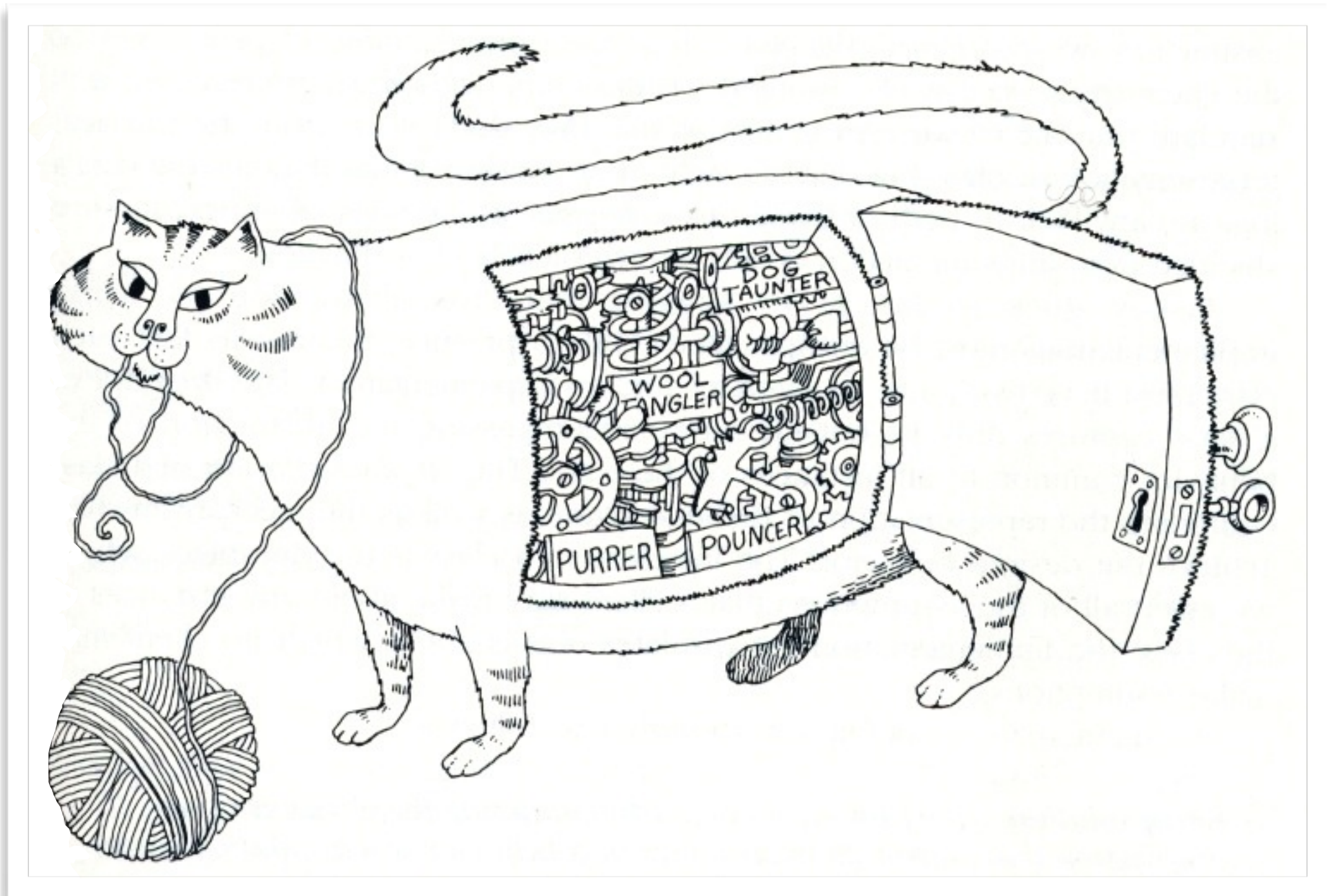
Encapsulation

- 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*

Encapsulation

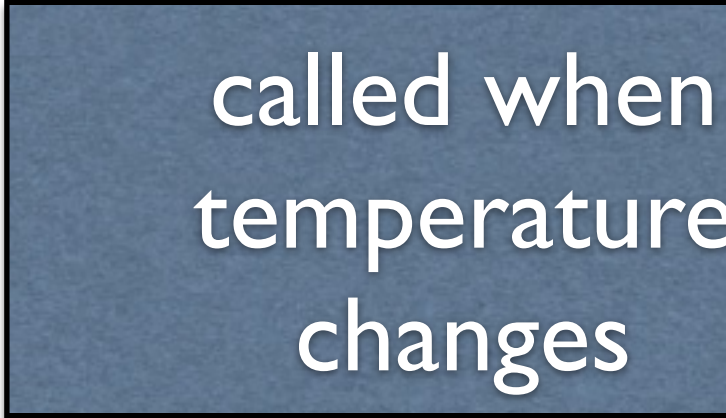
“Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and its behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.”

Encapsulation



An active Sensor

```
class ActiveSensor {  
public:  
    ActiveSensor(Location)  
    ~ActiveSensor();  
  
    void calibrate(Temperature actual);  
    Temperature currentTemperature() const;  
    Location location() const;  
  
    void register(void (*callback)(ActiveSensor *));  
  
private: ...  
}
```



called when
temperature
changes

Callback management is the sensor's secret

Anticipating Change

Features that are anticipated to 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;
```



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

```
const int ONE_HUNDRED = 100;  
int a[ONE_HUNDRED];
```

Number literals

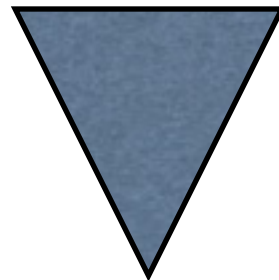
```
double sales_price = net_price * 1.19;
```



```
final double VAT = 1.19;  
double sales_price = net_price * VAT;
```

String literals

```
if (sensor.temperature() > 100)  
    printf("Water is boiling!");
```



```
if (sensor.temperature() > BOILING_POINT)  
    printf(message(BOILING_WARNING,  
                  "Water is boiling!"));
```

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


Principles

of object-oriented design

- Abstraction – Hide details
- Encapsulation – Keep changes local
- Modularity
- Hierarchy

Principles

of object-oriented design

- Abstraction – Hide details
- Encapsulation – Keep changes local
- **Modularity**
- Hierarchy

Modularity

- Basic idea: Partition a system such that parts can be designed and revised independently (“divide and conquer”)
- System is partitioned into *modules* that each fulfil a specific task
- Modules should be changeable and reuseable independent of other modules

Modularity



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 are in 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

- Modules should contain functions that logically belong together
- Achieved by grouping functions that work on the same data
- “natural” grouping in object oriented design

Weak coupling

- Changes in modules should not impact other modules
- Achieved via
 - Information hiding
 - 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*

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

- Reduces coupling between modules
- Disallow direct access to parts
- Limit the number of accessible classes
- Reduce dependencies
- Results in several new wrapper methods (“Demeter transmogrifiers”)

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

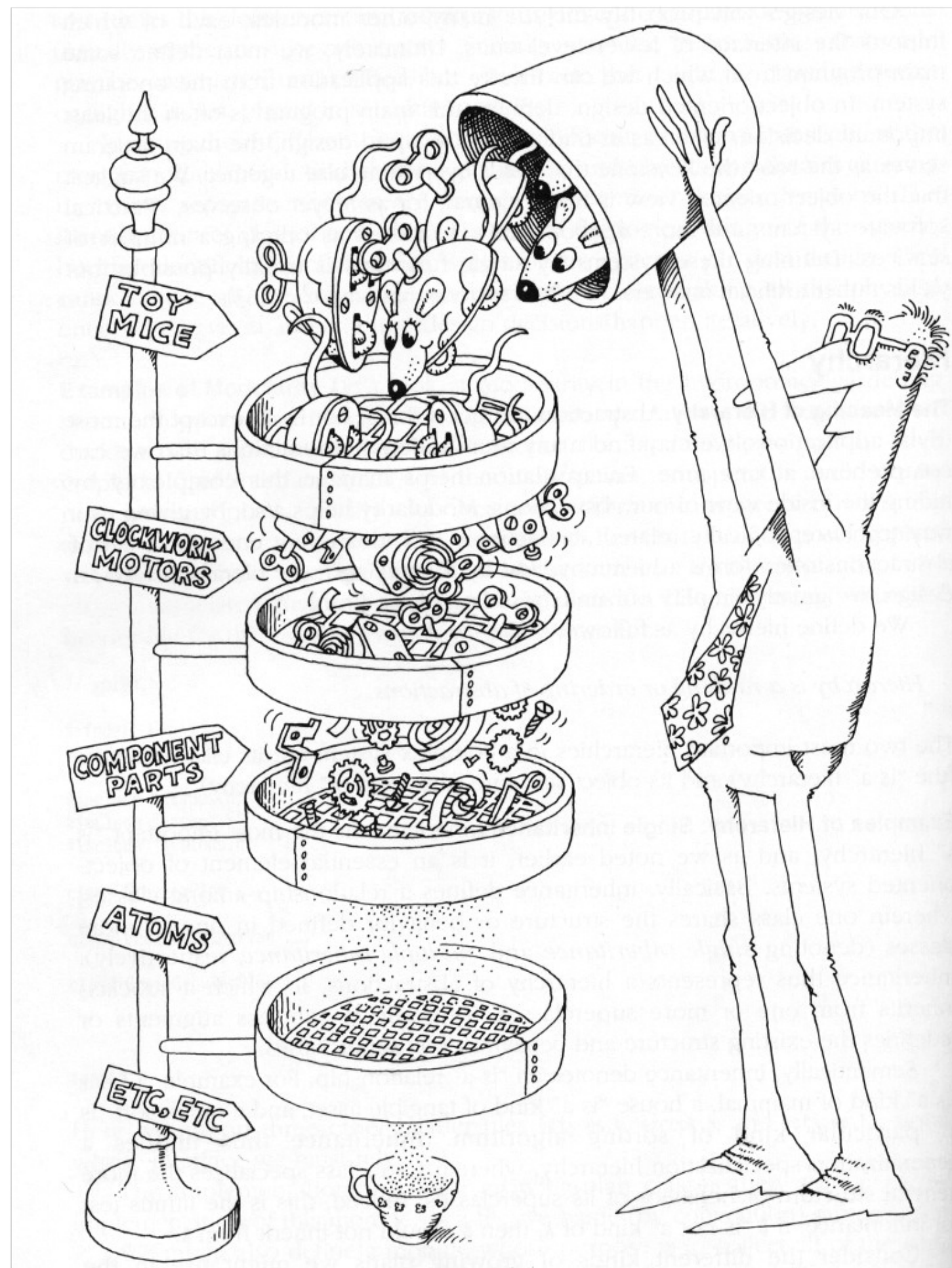
Principles

of object-oriented design

- Abstraction – Hide details
- Encapsulation – Keep changes local
- Modularity – Control information flow
High cohesion • weak coupling • talk only to friends
- **Hierarchy**

Hierarchy

“Hierarchy is a ranking or ordering of abstractions.”



Central Hierarchies

- “has-a” hierarchy –
Aggregation of abstractions
 - A car **has** three to four wheels
- “is-a” hierarchy –
Generalization across abstractions
 - An *ActiveSensor* **is a** *TemperatureSensor*

Central Hierarchies

- “has-a” hierarchy –
Aggregation of abstractions
 - *A car has three to four wheels*
- “is-a” hierarchy –
Generalization across abstractions
 - *An ActiveSensor is a TemperatureSensor*

Hierarchy principles

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

Hierarchy principles

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

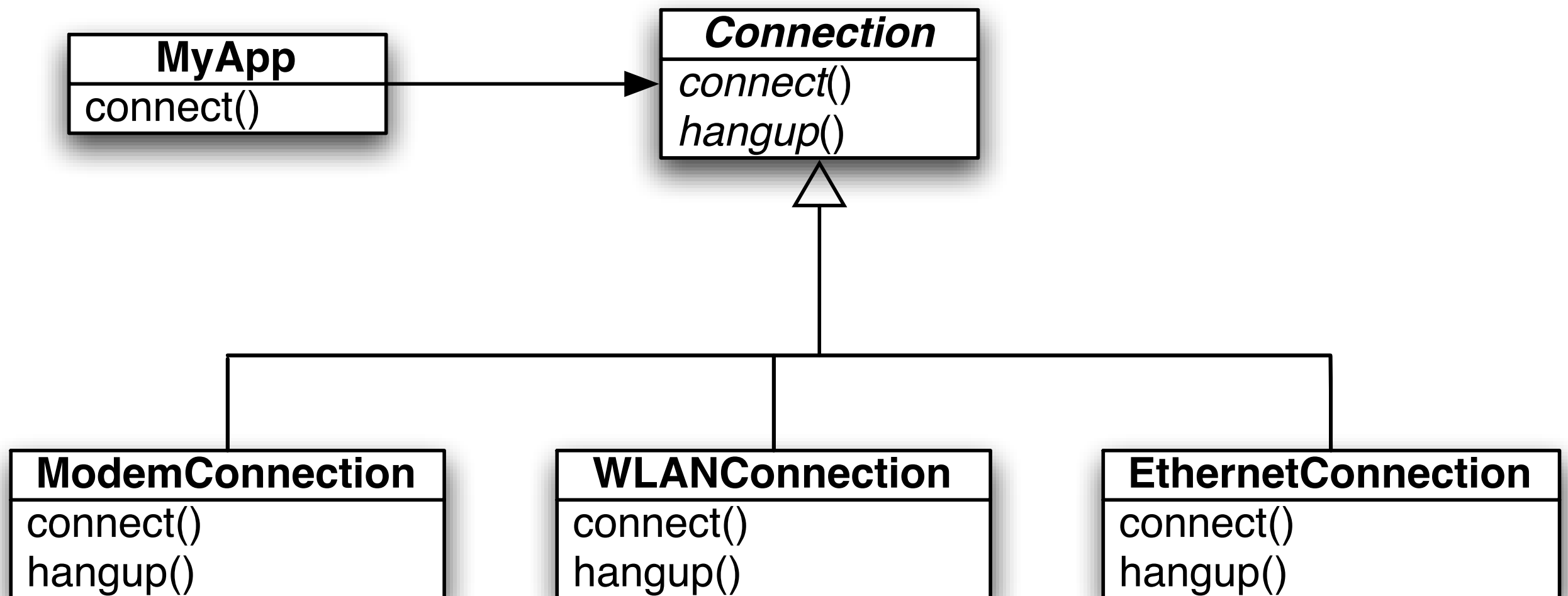
Open/Close principle

- 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) ...  
}
```

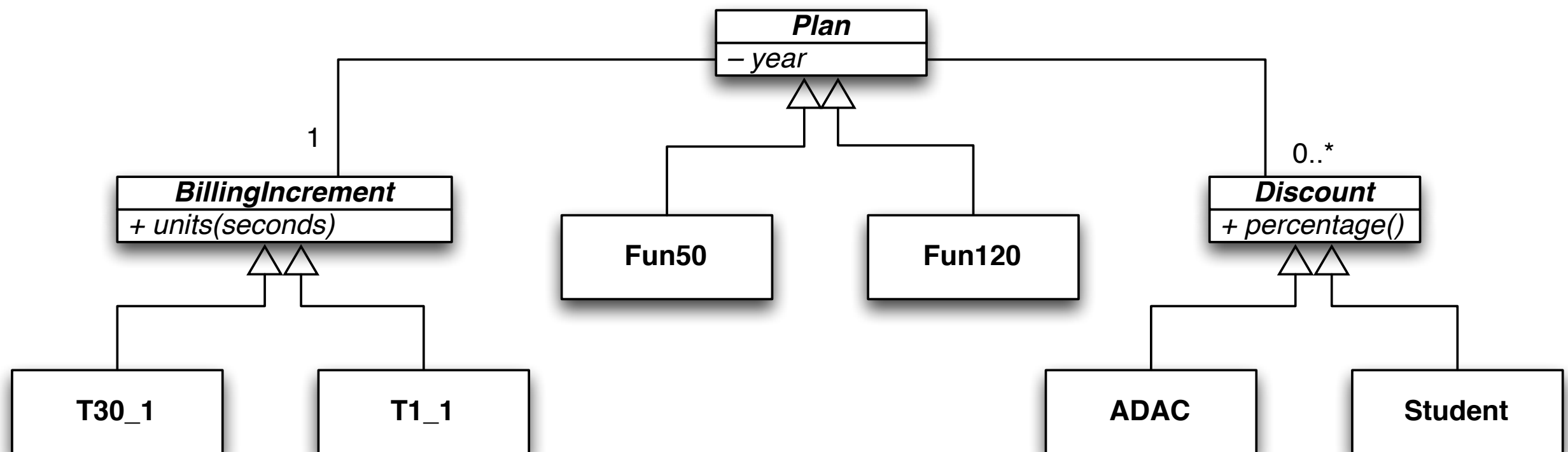
Solution with Hierarchies




```
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

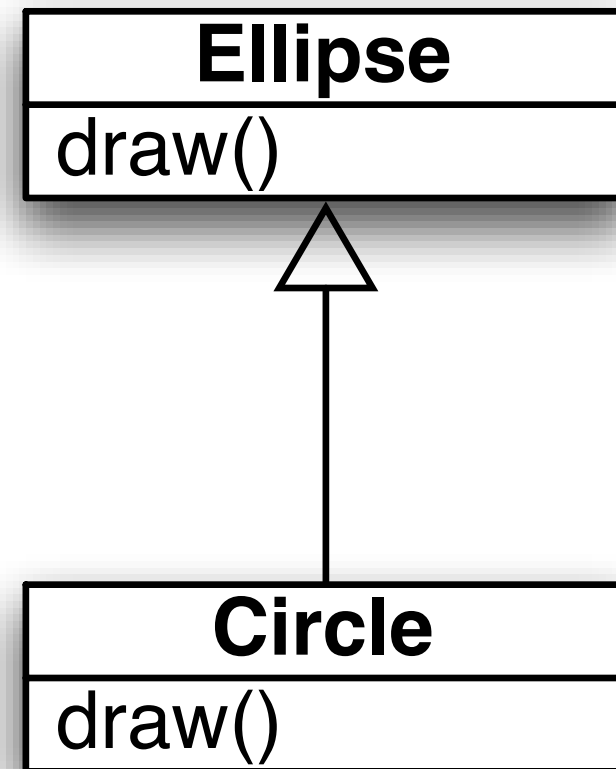
- Open/Close principle – Classes should be open for extensions
- Liskov principle – Subclasses should not require more, and not deliver less
- Dependency principle – Classes should only depend on abstractions

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 principle – Subclasses should not require more, and not deliver less
- **Dependency principle – Classes should only depend on abstractions**

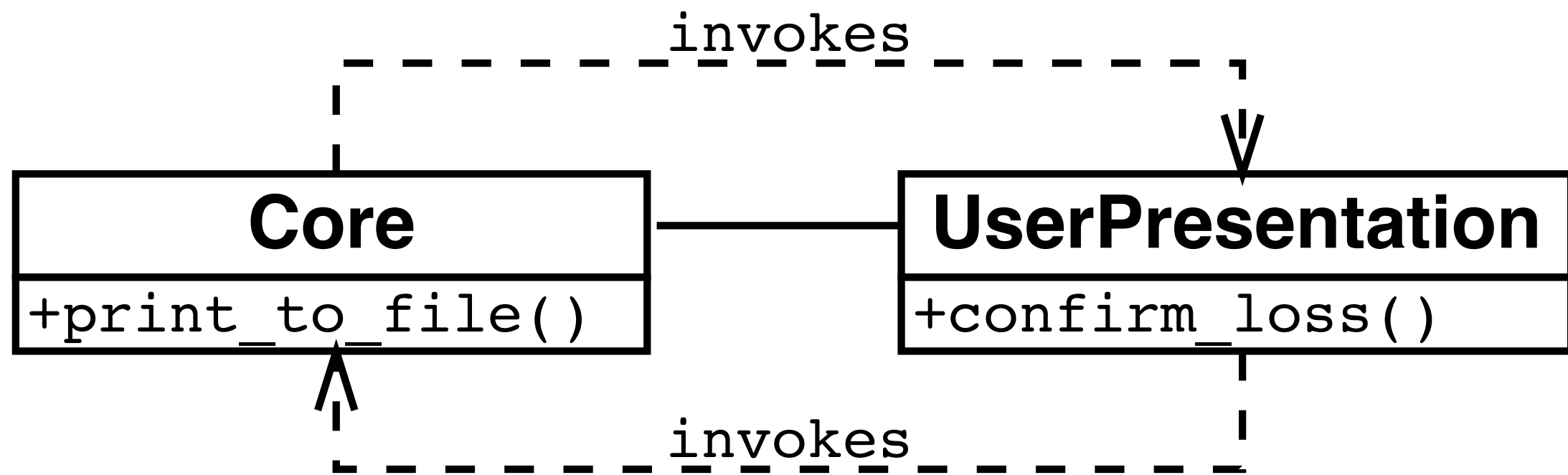
Dependency principle

- 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.  
void print_to_file(string filename)  
{  
    if (path_exists(filename))  
    {  
        // FILENAME exists;  
        // ask user to confirm overwrite  
        bool confirmed = confirm_loss(filename);  
        if (!confirmed)  
            return;  
    }  
  
    // Proceed printing to FILENAME  
    ...  
}
```


Cyclic Dependency

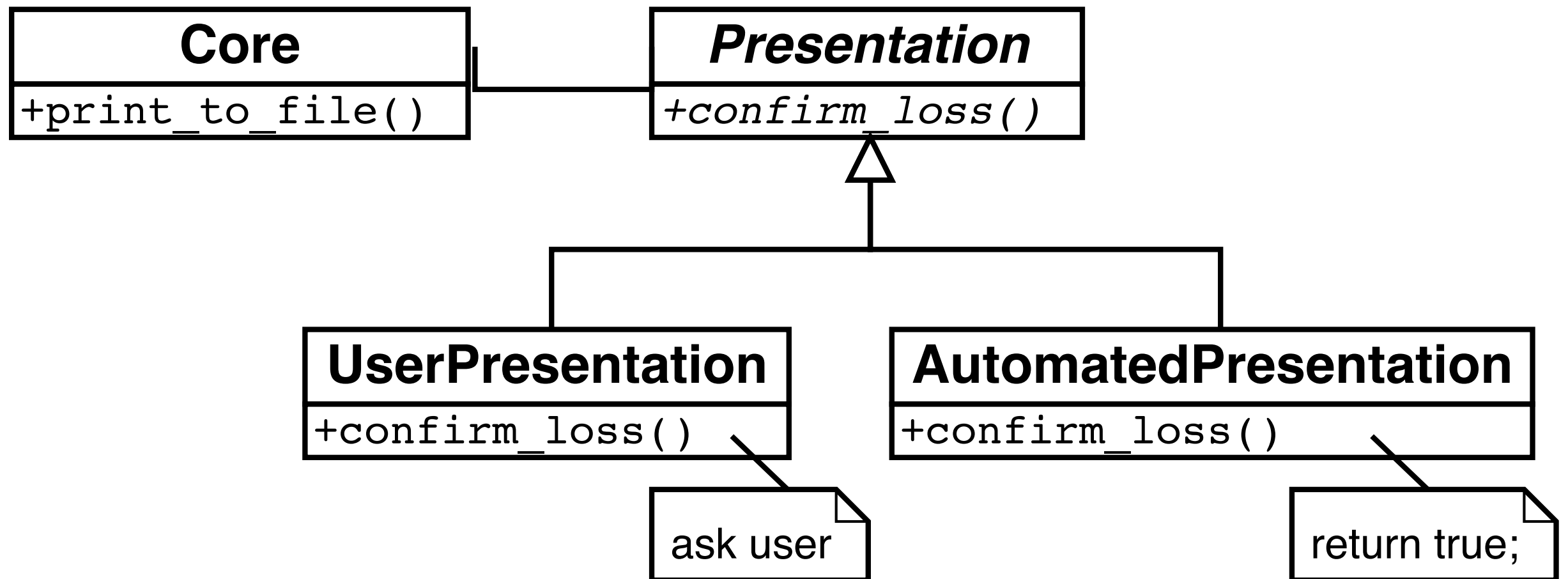


Constructing, testing, reusing individual modules becomes impossible!

Dependency

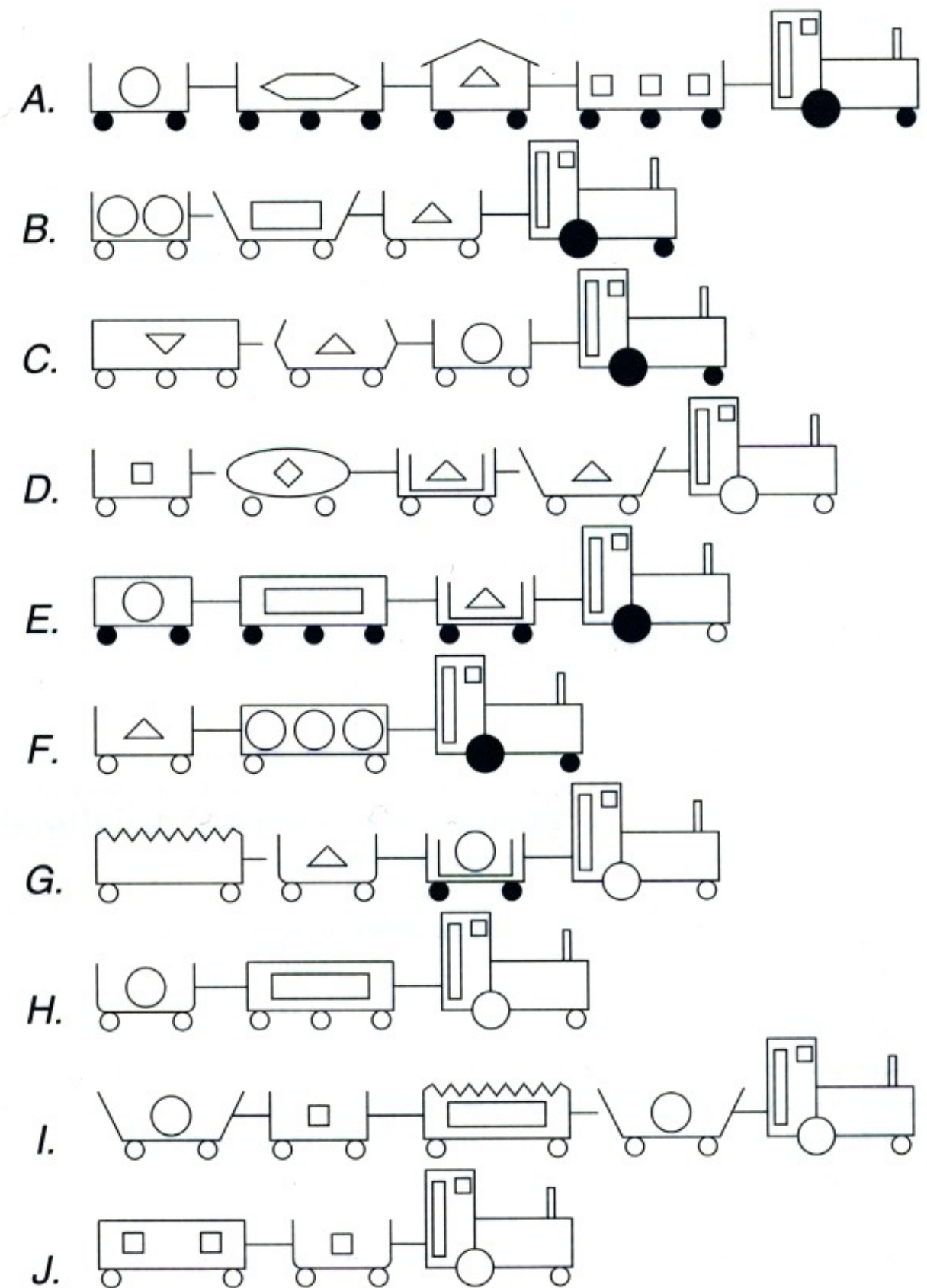
```
// Print current Web page to FILENAME.  
void print_to_file(string filename, Presentation *p)  
{  
    if (path_exists(filename))  
    {  
        // FILENAME exists;  
        // ask user to confirm overwrite  
        bool confirmed = p->confirm_loss(filename);  
        if (!confirmed)  
            return;  
    }  
  
    // Proceed printing to FILENAME  
    ...  
}
```

Depending on Abstraction



Choosing Abstraction

- Which is the “dominant” abstraction?
- How does this choice impact the remaining system?



Hierarchy principles

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

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
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

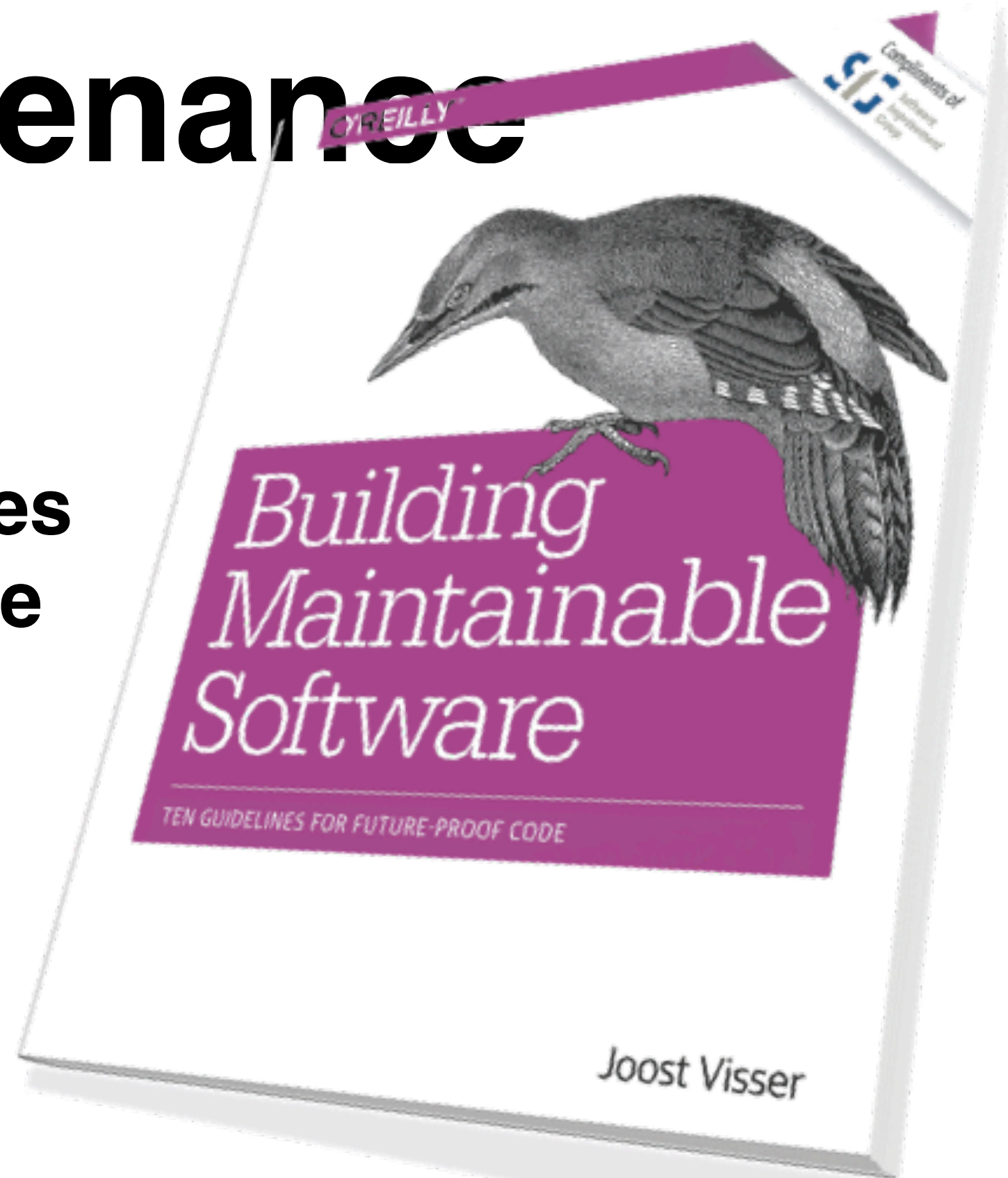
Goal: *Maintainability* and *Reusability*

Software Maintenance

- **Follow principles of object-oriented design**
- **Follow guidelines for maintainable software**

Software Maintenance

- Follow **guidelines for maintainable software**



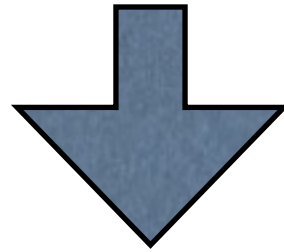
1. Write *Short* Units of Code

- Limit the length of code units to **~15 lines of code**
- Split long units into **multiple smaller units**
- Smaller units are **easier to understand**, easier to **test**, easier to **reuse**

Write *Short* Units of Code

```
public void doIt() {  
    // Query amount from database  
    results = executeQuery("SELECT account from "...);  
    if (results == null) ...  
  
    // Find account numbers  
    while (results.next()) {  
        ... whatever ...  
    }  
  
    // Store accounts in database again  
    ...  
}
```

Write *Short* Units of Code



```
public void doIt() {  
    results = queryAmounts();  
    accounts = findAccountNumbers(results);  
    storeAccounts(accounts);  
}
```

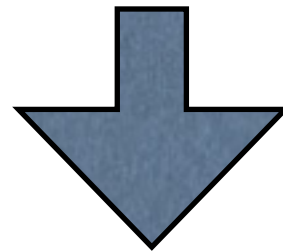
2. Write *Simple* Units of Code

- Limit the number of branches to 4
- Split complex units into simpler ones; avoiding complex units altogether
- Makes units easier to modify and test

Write *Simple* Units of Code

```
public List<Color> getFlagColors(Nationality nat) {  
    List<color> result;  
    switch (nat) {  
    case DUTCH:  
        result = Arrays.asList(RED, WHITE, BLUE);  
        break;  
    case GERMAN:  
        result = Arrays.asList(BLACK, RED, GOLD);  
        break;  
    case FRENCH:  
        result = Arrays.asList(BLUE, WHITE, RED);  
        break;  
    // ...  
}
```

Write *Simple* Units of Code



```
public class GermanNationality extends Nationality {  
    public List<Color> getFlagColors() {  
        return Array.asList(BLACK, RED, GOLD);  
    }  
}
```

3. Write Code Once

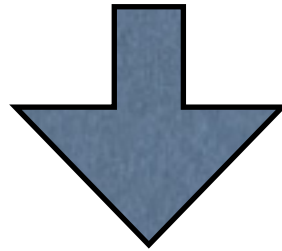
- **Do not copy code**
- **Write reusable, generic code and/or call existing methods**
- **When code is copied, bugs need to be fixed at multiple places**
- **Instead, introduce delegate method and/or superclass**

Write Code Once

```
public class CheckingAccount {  
    public Transfer makeTransfer(String account, Money amount) {  
        // Check account number  
        int sum = 0;  
        for (int i = 0; i < account.length(); i++)  
            sum += (9 - i) * convert(account.charAt(i));  
        if (sum % 11 != 0)  
            throw BusinessException("Invalid Account");  
        // ...  
    }  
}
```

```
public class SavingsAccount {  
    public Transfer makeTransfer(String account, Money amount) {  
        // Check account number  
        int sum = 0;  
        for (int i = 0; i < account.length(); i++)  
            sum += (9 - i) * convert(account.charAt(i));  
        if (sum % 11 != 0)  
            throw BusinessException("Invalid Account");  
        // ...  
    }  
}
```

Write Code Once



```
public class Account {
    public void validateAccount(String account) {
        int sum = 0;
        for (int i = 0; i < account.length(); i++)
            sum += (9 - i) * convert(account.charAt(i));
        if (sum % 11 != 0)
            throw BusinessException("Invalid Account");
    }
}

public class CheckingAccount extends Account {
    public Transfer makeTransfer(String account, Money amount) {
        validateAccount();
        // ...
    }
}

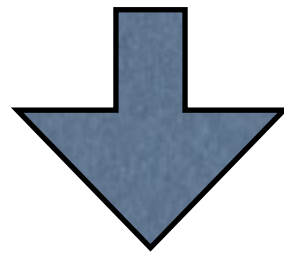
public class SavingsAccount extends Account {
    public Transfer makeTransfer(String account, Money amount) {
        validateAccount();
        // ...
    }
}
```

4. Keep Unit Interfaces Small

- Limit the number of parameters per unit to 4
- Do this by **extracting parameters into objects**
- Keeping the number of parameters low **makes units easier to understand and reuse**

Keep Unit Interfaces Small

```
private void drawRectangle(Canvas c, Graphics g, int x, int y, int w, int h)
```



```
class Rectangle {  
    private final Point position;  
    private final int width;  
    private final int height;  
    // ...  
    private void render(Canvas c, Graphics g);  
    // ...  
}
```

5. Separate Concerns In Modules

- **Avoid large modules to achieve loose coupling between them**
- **Do this by**
 - **assigning responsibilities to separate modules**
 - **hiding implementation details**
- **Changes in a loosely coupled code base are easier to oversee and execute**

6. Couple Architecture

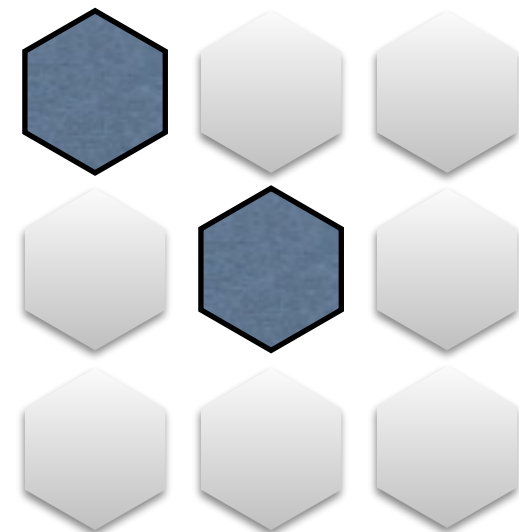
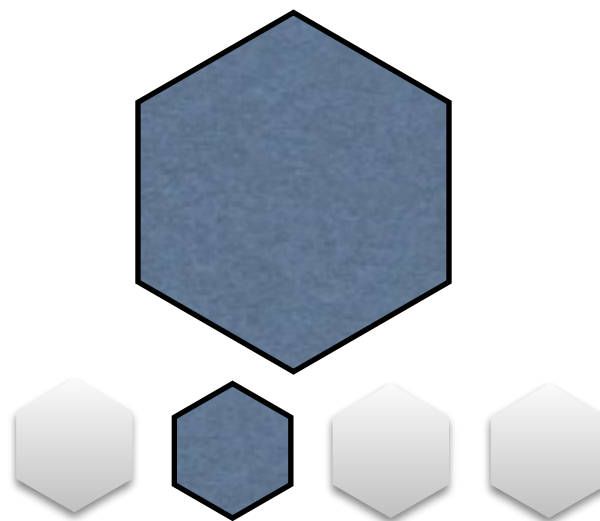
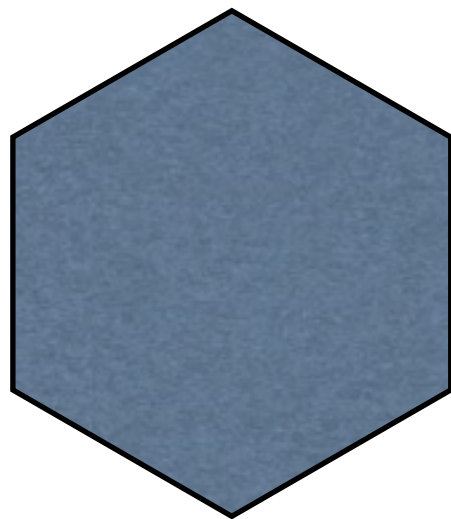
- **Achieve loose coupling between top-level components**
- **Do this by minimizing the amount of code that is exposed to modules in other components**
- **Independent components ease isolated maintenance**

7. Keep Architecture Components

- **Balance the number and relative size of top-level components in your code**
- **Do this by organizing the code in a way that the number of components is ~9 (between 6–12) and that the components are of approximately equal size**
- **Balanced components ease locating code and allow for isolated**

Keep Architecture Components

unbalanced  balanced



existing component



changes

8. Keep your Codebase Small

- **Keep your codebase as small as possible**
- **Do this by avoiding codebase growth and actively reducing system size**
- **A small product / project / team is a success factor**

9. Automate Tests

- **Automate tests for your codebase**
- **Do this by writing automated tests using a test framework**
- **Automated testing makes development predictable and less risky**

10. Write Clean Code

- **Write clean code.**
- Do this by **not leaving code smells behind** after development work
- Clean code is maintainable code

Write Clean Code

- Leave no **bad comments** behind
Whatever can be said in code needs no comment
- Leave no **code in comments** behind
Fix it or reject it
- Leave no **magic numbers** behind
 $\text{USD} = \text{EUR} * 1.09$. What could ever change?
- LeaveNoExcessivelyLongIdentifiersBehind()

Software Maintenance

- **Follow principles of object-oriented design**
- **Follow guidelines for maintainable software**