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

Imperative Programming

from 1950 until today
Programming Styles

- Chaotic
- Procedural
- Modular
- Object oriented

Chaos
Fortran • Algol (1954–1958)

Data

Programs sharing data – changes have *global effect*

Procedures
Fortran • Algol • Cobol • Lisp (1959–1961)

Data

Reusable subprograms with parameters
Modules
PL/I • Algol 68 • Pascal • Modula • Simula (1962–1970)

Data

Changes confined to individual modules

Gap

Objects
Smalltalk • C++ • Ada • Eiffel • Java (1980–)

Interface
State
Methods

Every object maintains its own state
Overview

<table>
<thead>
<tr>
<th>Generation</th>
<th>Control</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>chaotic</td>
<td>anything</td>
<td>anything</td>
</tr>
<tr>
<td>procedural</td>
<td>procedure</td>
<td>anything</td>
</tr>
<tr>
<td>modular</td>
<td>procedure</td>
<td>module</td>
</tr>
<tr>
<td>object oriented</td>
<td>method</td>
<td>object</td>
</tr>
</tbody>
</table>

plus: logic-based, rule-based, constraint-based, functional programming…

Principles
of object-oriented design

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

Goal: Maintainability and Reusability
Abstraction

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

“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”
An Engineer’s Solution

```c
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: ...
}

More Abstraction

Leci 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: …
}

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

```c
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;
```

Number literals

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

String literals

```c
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

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 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
Open/Close principle

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

An Internet Connection

```java
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

![Diagram showing the hierarchy of classes and methods related to connecting to the internet.](image)
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

http://en.wikipedia.org/wiki/Liskov_substitution_principle
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

// 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
    ...
}
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

More on this topic: aspect-oriented programming
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