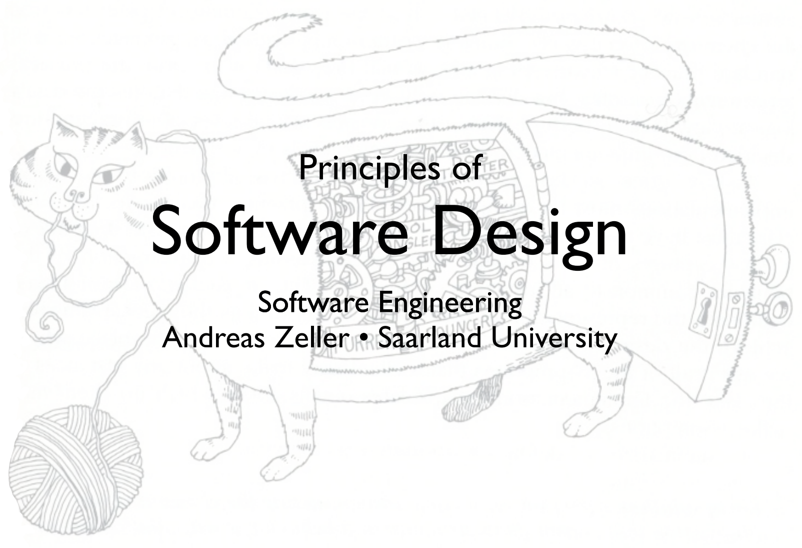


Principles of

Software Design

Software Engineering
Andreas Zeller • Saarland University

These slides are based on Grady Booch: Object-Oriented Analysis and Design (1998), updated from various sources

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

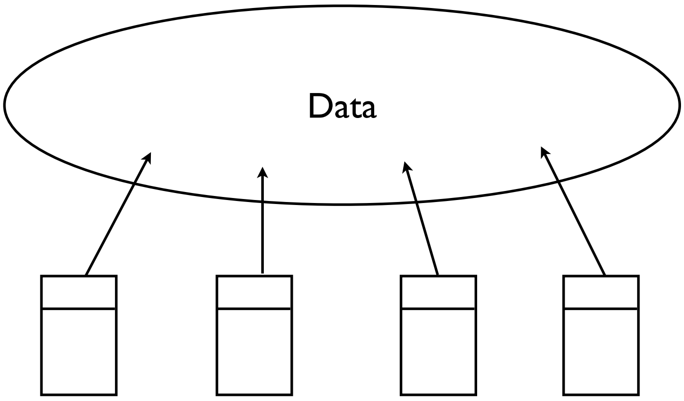
from 1950 until today

Programming Styles

- Chaotic
- Procedural
- Modular
- Object oriented

Chaos

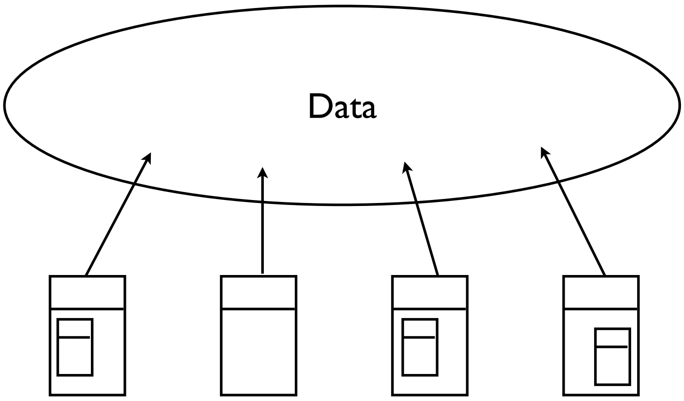
Fortran • Algol (1954–1958)



Programs sharing data – changes have *global effect*

Procedures

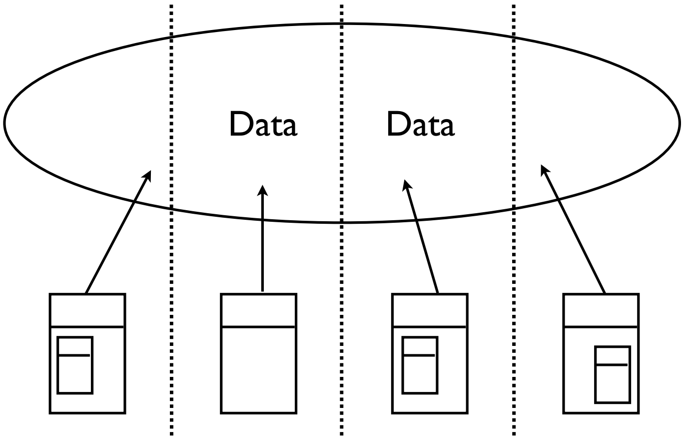
Fortran • Algol • Cobol • Lisp (1959–1961)



Reusable subprograms with parameters

Modules

PL/I • Algol 68 • Pascal • Modula • Simula (1962–1970)



Changes *confined to individual modules*

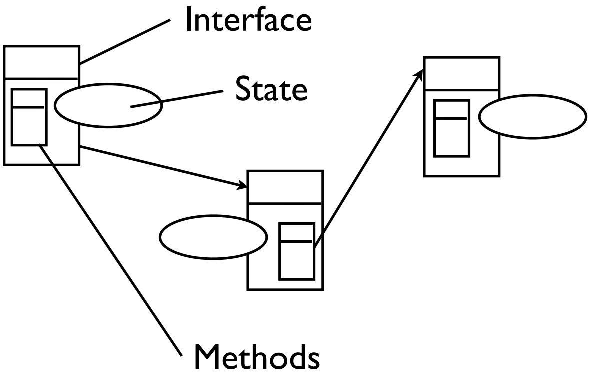
Gap

(1970–1980)

?

Objects

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



Every object maintains its own state

Overview

Generation	Control	Data
chaotic	anything	anything
procedural	procedure	anything
modular	procedure	module
object oriented	method	object

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

Principles

of object-oriented design

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

Goal: *Maintainability* and *Reusability*

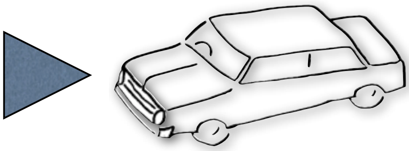
Principles

of object-oriented design

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

Abstraction

Concrete Object



General Principle

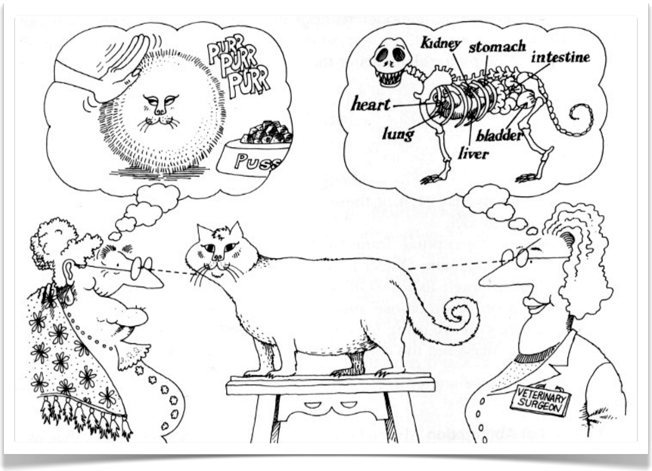
Abstraction…

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

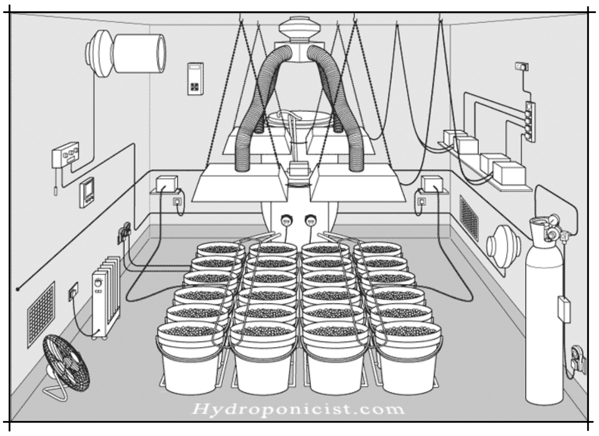
Abstraction

“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

Principles

of object-oriented design

- Abstraction – hide details
- Encapsulation
- Modularity
- Hierarchy

Principles

of object-oriented design

- Abstraction – Hide details
- Encapsulation
- Modularity
- Hierarchy

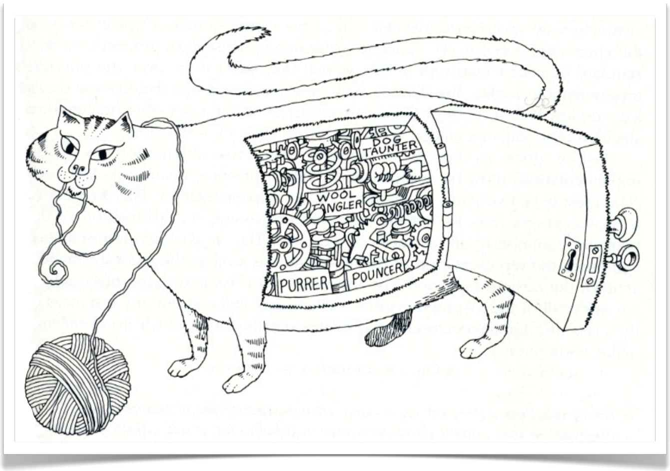
Encapsulation

- 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];~~
```

If one searches for “100”, one will miss the “99” :-(

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

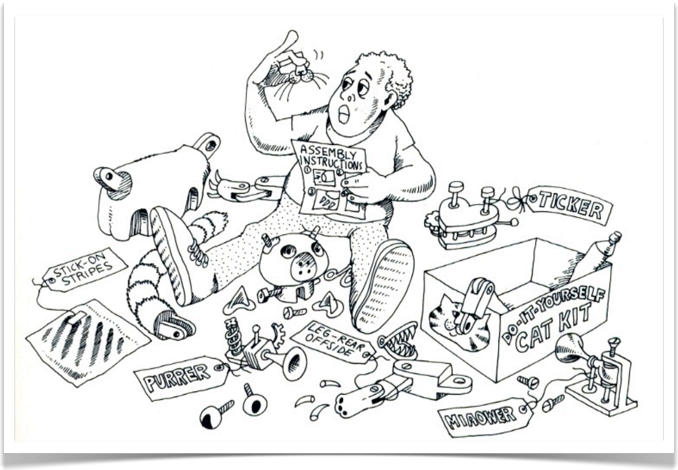
of object-oriented design

- 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*

Demeter = Greek Goddess of Agriculture; grow software in small steps; signify a bottom-up philosophy of programming

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"

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

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

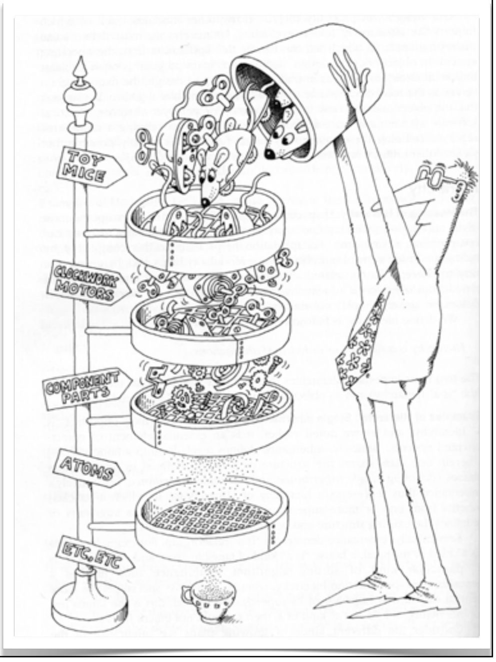
- Abstraction – Hide details
- Encapsulation – Keep changes local
- Modularity – Control information flow
  High cohesion • weak coupling • talk only to friends
- Hierarchy

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

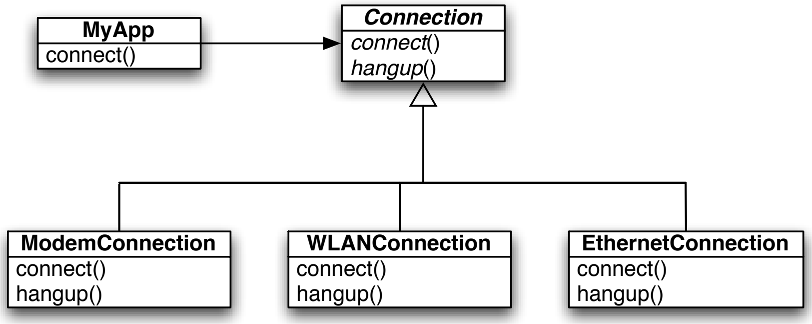
Open/Close principle

- 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



MyApp
connect()

Connection
connect()
hangup()

ModemConnection
connect()
hangup()

WLANConnection
connect()
hangup()

EthernetConnection
connect()
hangup()

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

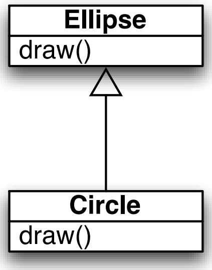
Liskov Substitution Principle

- 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*

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

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

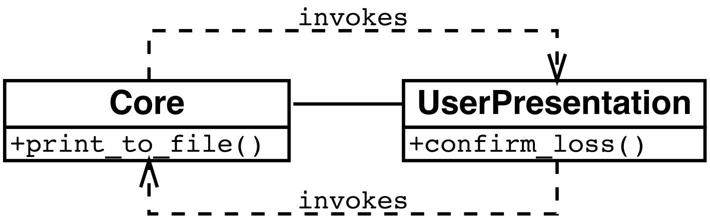
Dependency principle

- 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

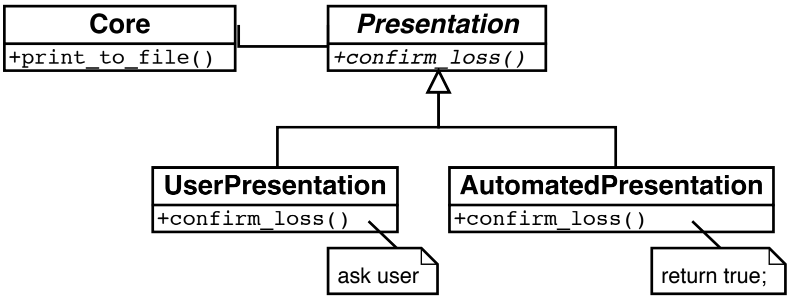


Constructing, testing, reusing individual modules becomes impossible!

Dependency

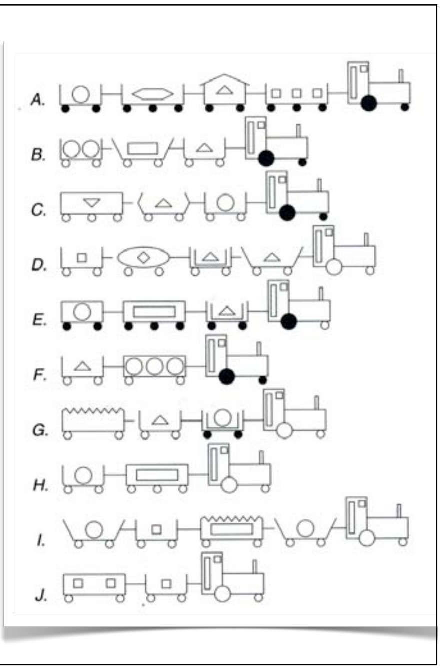
```
// Print current Web page to FILENAME.
void print_to_file(string filename, Presentation *p)
{
    if (path_exists(filename))
    {
        // FILENAME exists;
        // ask user to confirm overwrite
        bool confirmed = p->confirm_loss(filename);
        if (!confirmed)
            return;
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

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

Depending on Abstraction



Choosing Abstraction

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



More on this topic: aspect-oriented programming

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*

From Requirements to Software Design

Software Engineering
Andreas Zeller • Saarland University

These slides are based on Grady Booch: Object-Oriented Analysis and Design (1998), updated from various sources

---

From Requirements to Software Design

- Describe requirements as *use cases*
- Refine use cases to *alternate scenarios*
- Identify *classes* and *operations*

See Pressman, chapter 8 for the remainder of this lecture

Use Case

- An *actor* is something that can act – a person, a system, or an organization
- A *scenario* is a specific sequence of *actions* and *interactions* between actors (where at least one actor is a system)
- A *use case* is a collection of related scenarios – successful and failing ones

Actors and Goals

- What are the *boundaries* of the system? Is it the software, hardware and software, also the user, or a whole organization?
- Who are the *primary actors* – i.e., the stakeholders?
- What are the *goals* of these actors?
- Describe how the system fulfills these goals (including all exceptions)

Example: SafeHome

Initial Scenario

Use case: *display camera views*
Actor: *homeowner*

If I'm at a remote location, I can use any PC with appropriate browser software to log on to the SafeHome Web site. I enter my user ID and two levels of passwords and, once I'm validated, I have access to all the functionality. To access a specific camera view, I select "surveillance" and then "select a camera". Alternatively, I can look at thumbnail snapshots from all cameras by selecting "all cameras". Once I choose a camera, I select "view"…

Refined Scenario

Use case: *display camera views*
Actor: *homeowner*

1. *The homeowner logs on to the Web Site*
2. *The homeowner enters his/her user ID*
3. *The homeowner enters two passwords*
4. *The system displays all major function buttons*
5. *The homeowner selects "surveillance" button*
6. *The homeowner selects "Pick a camera"…*

Alternative Interactions

- Can the actor take some other action at this point?
- Is it possible that the actor encounters some error condition? If so, which one?
- Is it possible that some other behavior is encountered? If so, which one?

SAFEHOME

Use-Case Template for Surveillance

Use-case: Access camera surveillance—display camera views (ACS-DCV).

Primary actor: Homeowner.

Goal in context: To view output of camera placed throughout the house from any remote location via the Internet.

Preconditions: System must be fully configured; appropriate user ID and passwords must be obtained.

Trigger: The homeowner decides to take a look inside the house while away.

Scenario:

1. The homeowner logs onto the *SafeHome Products* Web site.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects "surveillance" from the major function buttons.
6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the "view" button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

Exceptions:

1. ID or passwords are incorrect or not recognized—see use-case: "validate ID and passwords."
2. Surveillance function not configured for this system—system displays appropriate error message; see use-case: "configure surveillance function."
3. Homeowner selects "view thumbnail snapshots for all cameras"—see use-case: "view thumbnail snapshots for all cameras."
4. A floor plan is not available or has not been configured—display appropriate error message and see use-case: "configure floor plan."
5. An alarm condition is encountered—see use-case: "alarm condition encountered."

Priority: Moderate priority, to be implemented after basic functions.

When available: Third increment.

Frequency of use: Infrequent.

SAFEHOME

Use-Case Template for Surv

Use-case: Access camera surveillance—display camera views (ACS-DCV).

Primary actor: Homeowner.

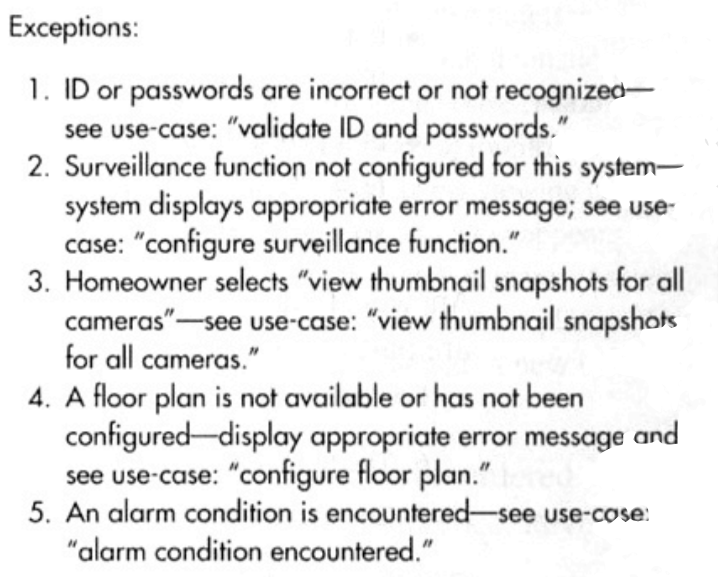
Goal in context: To view output of camera placed throughout the house from any remote location via the Internet.

Preconditions: System must be fully configured; appropriate user ID and passwords must be obtained.

Trigger: The homeowner decides to take a look inside the house while away.

Scenario:

1. The homeowner logs onto the *SafeHome Products* Web site.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects "surveillance" from the major function buttons.
6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the "view" button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

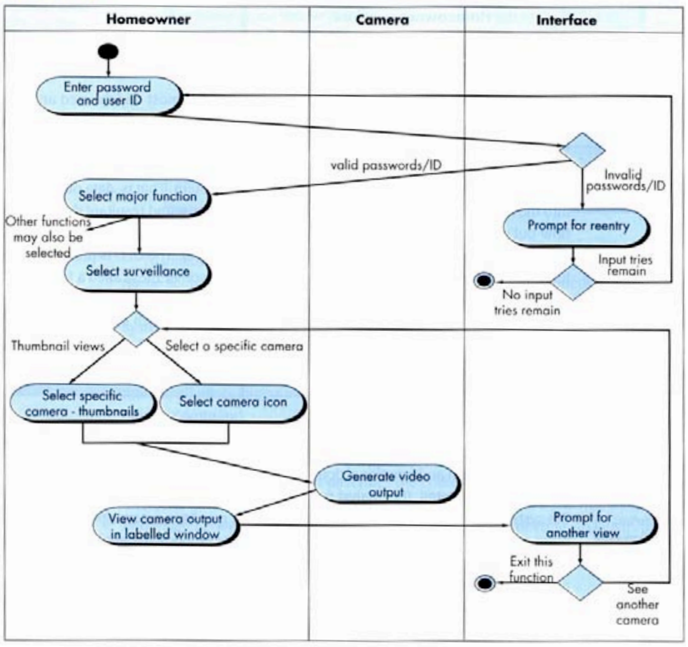


Exceptions:

1. ID or passwords are incorrect or not recognized—see use-case: "validate ID and passwords."
2. Surveillance function not configured for this system—system displays appropriate error message; see use-case: "configure surveillance function."
3. Homeowner selects "view thumbnail snapshots for all cameras"—see use-case: "view thumbnail snapshots for all cameras."
4. A floor plan is not available or has not been configured—display appropriate error message and see use-case: "configure floor plan."
5. An alarm condition is encountered—see use-case: "alarm condition encountered."

From Use Case to Control

- To describe the *flow of interaction* (and possible errors / exceptions), one uses an *activity diagram*.
- The activity diagram represents the interaction flow through the system
- Useful *swimlane* variant: arranged according to actors



Swimlane diagram for Access camera surveillance–display camera views functions

Class-based modeling

Initial approach:

- Each *noun* in the problem description becomes a class candidate
- *Verbs* later become methods
- A class should never have an imperative procedural name (such as *InvertImage*)

Requirements for

Potential Classes

1. Retained Information
   The information is necessary for the system to function
2. Needed Services
   The potential class must have a set of potential operations
3. Multiple Attributes
   We are focusing on potential classes with more than one attribute
4. Common Attributes and Operations
   The attributes and operations apply to all instances of the class
5. Essential Requirements
   External entities – producers and consumers of information – almost always become classes

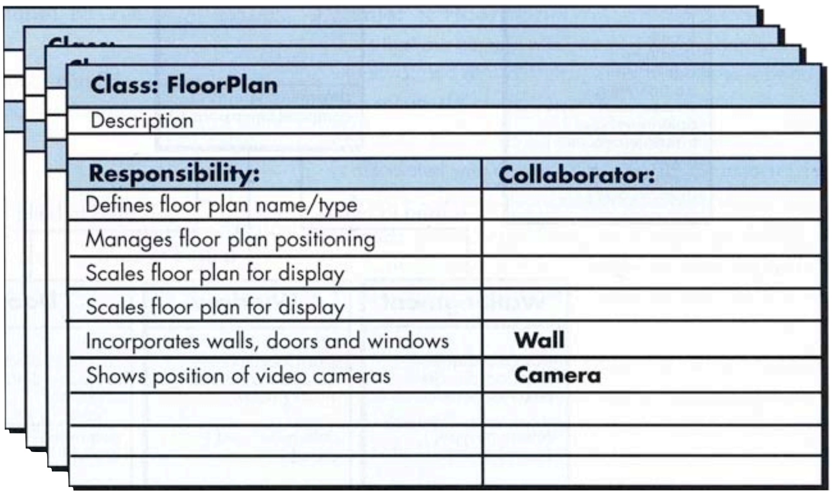
These are requirements a potential class has to fulfill to be retained

---

Classes and Methods

- *Class-Responsibility-Collaborator (CRC) modeling* is a simple means for identifying and organizing classes
- Makes use of virtual or actual *index cards*

A CRC index card



Class: FloorPlan

Description

Responsibility:	**Collaborator:**
Defines floor plan name/type	
Manages floor plan positioning	
Scales floor plan for display	
Scales floor plan for display	
Incorporates walls, doors and windows	**Wall**
Shows position of video cameras	**Camera**

CRC Responsibilities

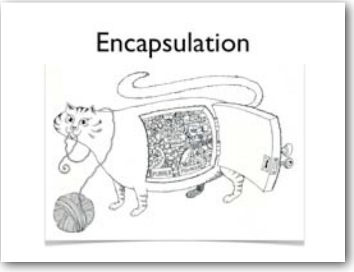
- System intelligence should be *distributed across classes* (➔ modularity)
- State responsibilities *as general as possible* (➔ abstraction)
- Information and related behavior goes into the same class (➔ encapsulation)
- Information about one thing should be *localized in a single class* (➔ modularity)
- Responsibilities should be *shared among related classes* (➔ hierarchy)

CRC Collaborations

- If a class cannot fulfil a responsibility, it has to *collaborate with other classes.*
- Typical (generic) relationships include
  - *is-part-of* – parts of an aggregate class
  - *has-knowledge-of* – information source
  - *depends-upon* – required for existence

Final word on CRC

“One purpose of CRC cards is to fail early, to fail often, and to fail inexpensively. It is a lot cheaper to tear up a bunch of cards than it would be to reorganize a large amount of source code.” *(C. Horstmann)*



Encapsulation

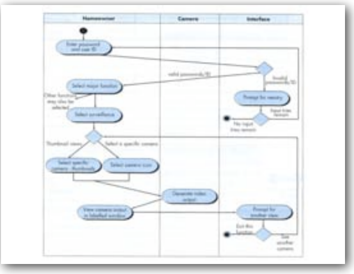
Principles

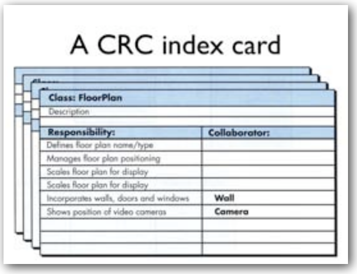
of object-oriented design

- 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*

Summary





A CRC index card

Class: FloorPlan	
Description	
Responsibility:	**Collaborator:**
Defines floor plan name/type	
Manages floor plan positioning	
Scales floor plan for display	
Scales floor plan for display	
Incorporates walls, doors and windows	**Wall**
Shows position of video cameras	**Camera**