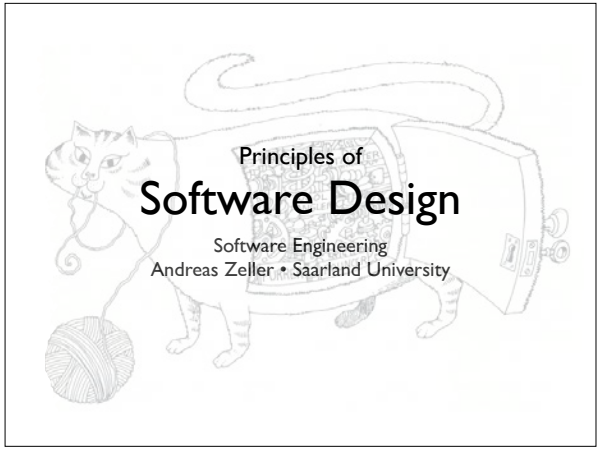


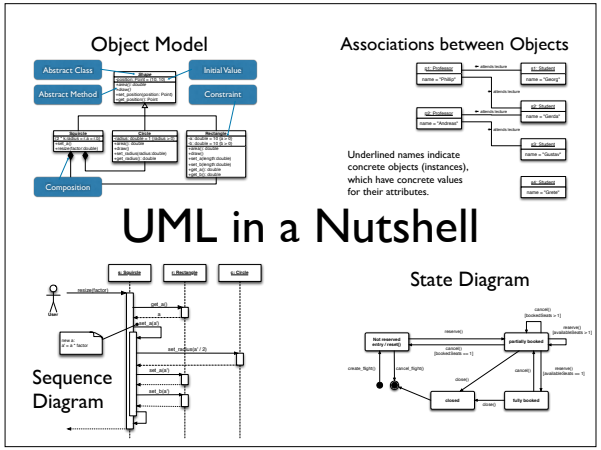
1

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

2



3

UML Recap

- Want a *notation* to express OO designs
- UML = *Unified Modeling Language*
- a standardized (ISO/IEC 19501:2005), *general-purpose modeling language*
- includes a set of *graphic notation techniques* to create visual models of *object-oriented* software-intensive systems

UML Creators


Grady Booch


Jim Rumbaugh

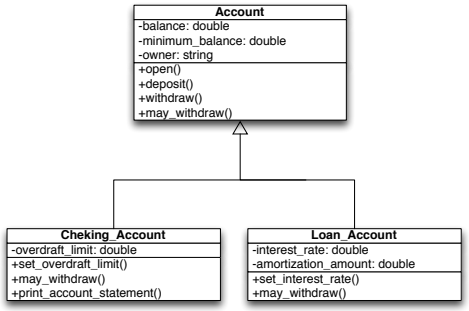

Ivar Jacobson

Object-Oriented Modeling in UML

includes the following design aspects:

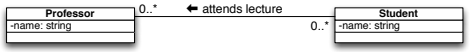
- *Object model*: *Which* objects do we need?
  - Which are the *features* of these objects? (attributes, methods)
  - How can these objects be *classified*? (Class hierarchy)
  - What *associations* are there between the classes?
- *Sequence diagram*: How do the objects *act together*?
- *State chart:* What *states* are the objects in?

Object Model



7

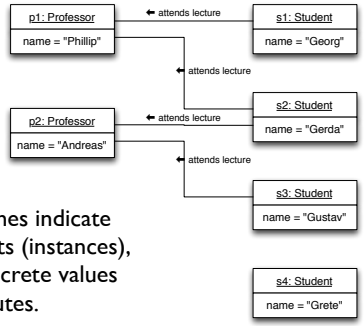
Associations



Professors have multiple students,
and students have multiple professors.

8

Associations between Objects

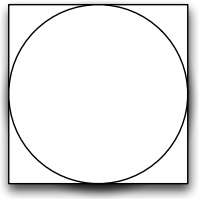


Underlined names indicate
concrete objects (instances),
which have concrete values
for their attributes.

9

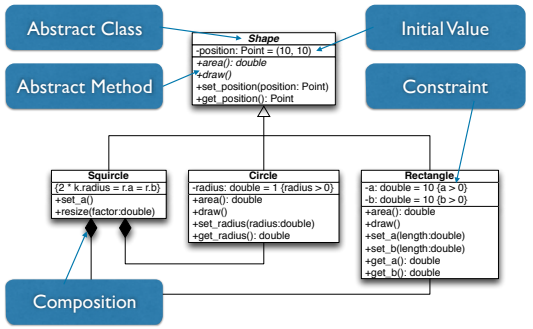
Composition

A "squircle" consists of a circle on top of a square:



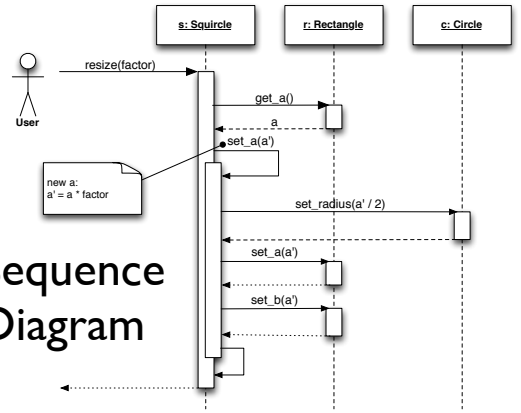
10

Composition



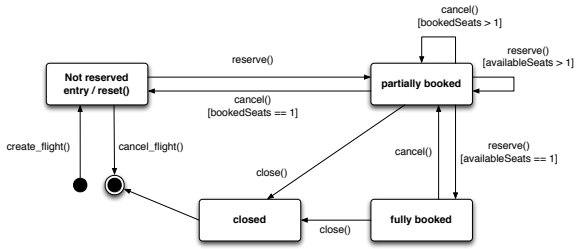
11

Sequence Diagram



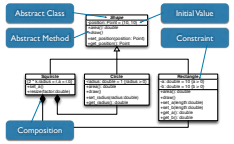
12

State Diagram

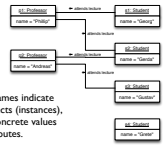


13

Object Model

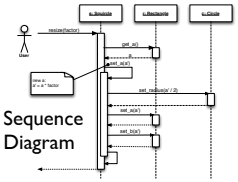


Associations between Objects



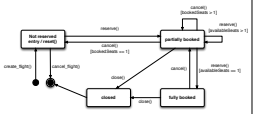
Underlined names indicate concrete objects (instances), which have concrete values for their attributes.

UML in a Nutshell

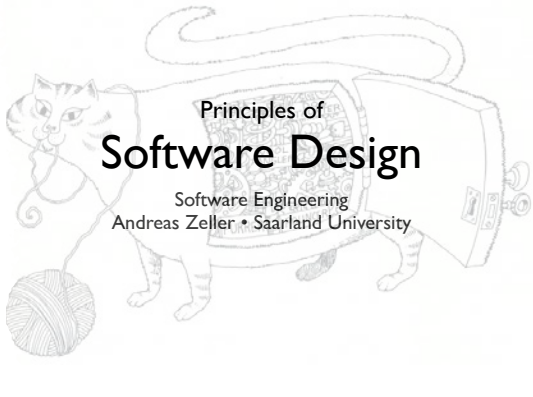


Sequence Diagram

State Diagram



14



15

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

Principles
of object-oriented design

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

Goal: *Maintainability* and *Reusability*

16

Principles
of object-oriented design

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

17

Abstraction

Concrete Object

General Principle

18

Abstraction…

19

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

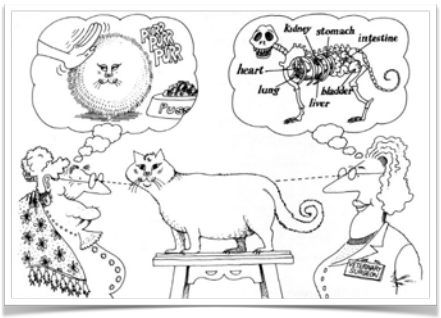
Abstraction

20

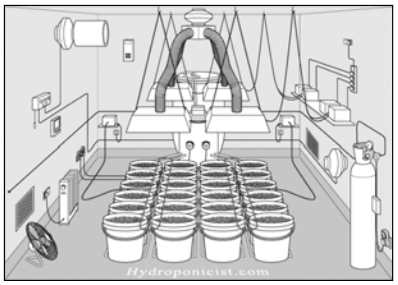
"An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer"

Perspectives

21



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

25

Principles
of object-oriented design

- Abstraction – hide details
- Encapsulation
- Modularity
- Hierarchy

26

Principles
of object-oriented design

- Abstraction – Hide details
- Encapsulation
- Modularity
- Hierarchy

27

28

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*

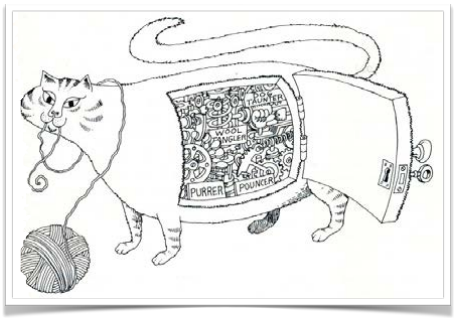
29

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."

30

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

31

Anticipating Change

Features that are anticipated to change should be *isolated* in specific components

- Number literals
- String literals
- Presentation and interaction

32

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];~~

33

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;
```

34

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();
```

35

Principles

of object-oriented design

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

36

Principles
of object-oriented design

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

37

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

38

Modularity

39

Modularity

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

40

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

41

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

42

High cohesion

43

- 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

44

- Changes in modules should not impact other modules
- Achieved via
  - Information hiding
  - Depending on as few modules as possible

Law of Demeter

45

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

46

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

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

47

```
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

48

```
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")

49

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

50

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

51

Hierarchy

"Hierarchy is a ranking or ordering of abstractions."

52

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*

53

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*

54

Hierarchy principles

55

- 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

56

- 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

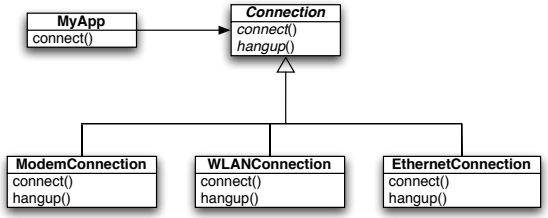
57

- 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



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

61

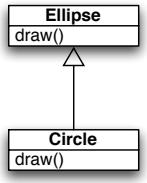
http://en.wikipedia.org/wiki/Liskov_substitution_principle

- 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

62

- Every circle is an ellipse
- Does this hierarchy make sense?
- No, as a circle *requires more* and *delivers less*



Hierarchy principles

63

- 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

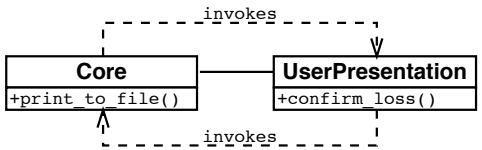
- A class should only depend on *abstractions* – never on concrete subclasses *(dependency inversion principle)*
- This principle can be used to *break* dependencies

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!

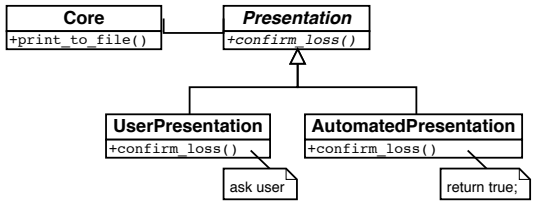
Dependency

```
// 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
    ...
}
```

67

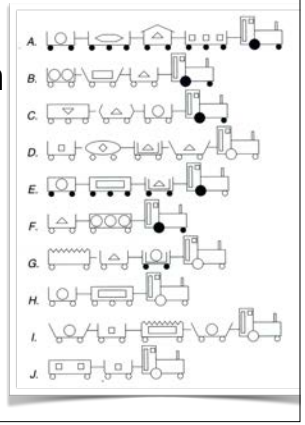
Depending on Abstraction



68

Choosing Abstraction

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



69

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

70

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

71

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*

72