Programming II

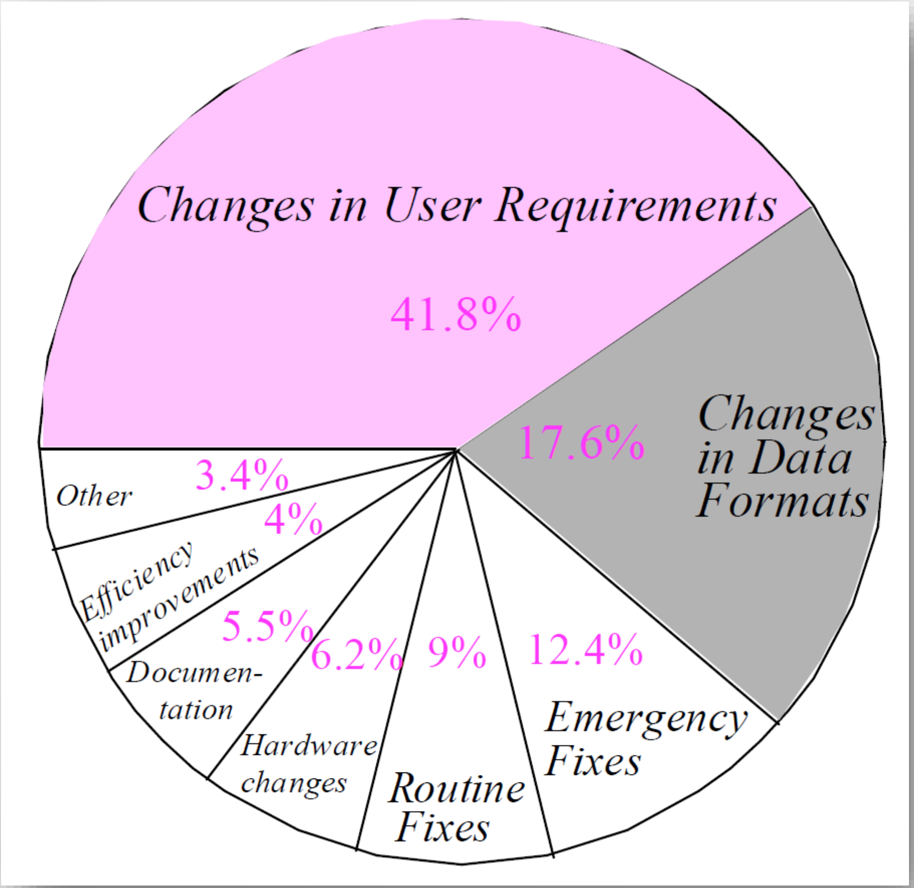
Object Orientation

Lecture Outline

- Concept of object-oriented design (why we need objects)
- Class
- Example

Why we need objects…

Software Maintenance



- Study of Lientz and Swanson (1980)
- 487 information systems
- **How much effort does it require?**

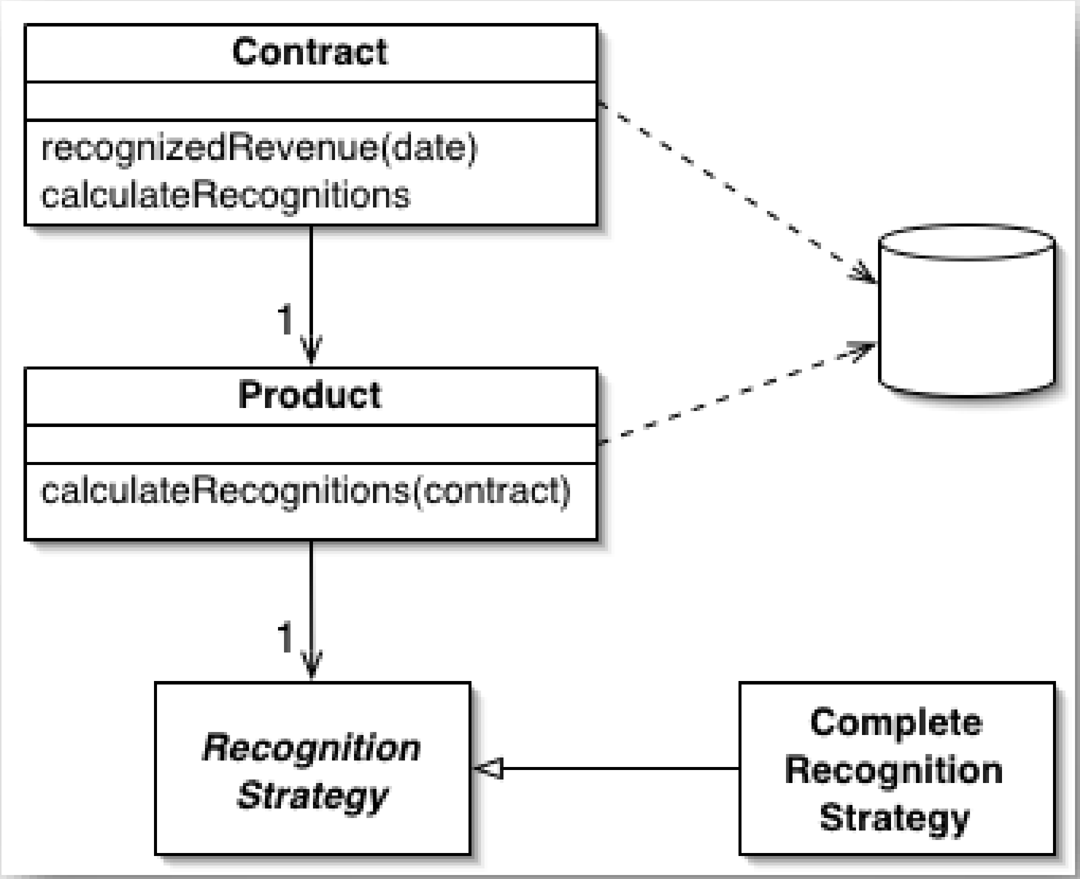
Object Orientation

- Object-oriented techniques help write software with better maintainability
  - Method and language
  - Implementation and environment
  - Libraries

Method and Language

- It is not just programming language and how to use it.
- It is also a way of thinking and expressing ...
- ... and also about records in textual or graphical form.

Domain Model



Implementation and Language

- Support for the development
  - Features and efficiency of development tools
  - Tools to support the deployment of new versions
  - Tools to support documentation

Libraries

- Object-oriented technologies rely heavily on reusability.
  - To support development by using of already implemented solutions (libraries)
  - It is also to support the creation and management of new custom libraries

Should we be dogmatic?

- The object-oriented approach is an essential tool for software development. However…
  - There are various programming languages with varying degrees of support for object-oriented programming techniques (OOP).
  - Not everyone needs all the features that OOP offers.
  - Object orientation may be just one factor in the successful development, and therefore, it should be considered comprehensively.

Method and Language

- Classes
- Classes as modules
- Classes as types
- Message passing (feature call)
- Information hiding
- The static type checking
- Genericity
- Inheritance, redefinition, polymorphism and dynamic binding
- Memory management and garbage collection

Class

Classes

> The method and the language should have the notion of class as their central concept.

- The object-oriented approach is based on the term class.
- The class can be seen as part of the software, which describes the abstract data type and its implementation.
- As the abstract data type, we understand a set of objects with a common behavior represented by a list of operations that objects can operate.

Classes as Modules

Classes should be the only modules.

- The OOP is mainly about the structure (architecture) software; its priority is modularity.
- Classes not only describe the types of objects; they must also be modular units.
- In pure object-oriented programs should not be other separate units than classes (e.g.,. functions).

Classes as Types

Every type should be based on a class.

- In pure object-oriented languages and programs should not be other types than the classes.

- This principle also can be applied to the system types such as INT or FLOAT.

Message Passing

Feature call should be the primary computational mechanism.

- Message Passing (feature class), feature-based computation - a computational mechanism.
  - A named message with parameters is sent to an object (an instance of a class).
  - *aPerson-> ChangeLastName ("Smith")*
  - Whoever sends a message (requesting execution of operations with certain arguments) is a CLIENT of the class.

Information Hiding

It should be possible for the author of a class to specify that a feature is available to all clients, to no client, or to specified clients.

- For the client are important only the operations (methods) that describe the external behavior of objects.
- Details of implementation should be hidden (data + private operations).
- If a client needs to obtain information about the state (data) of an object, it is possible only by sending a message.

Static Binding a Type Checking

> A well-defined type system should, by enforcing a number of type declaration and compatibility rules, guarantee the run-time type safety of the systems it accepts.

- Each entity in the program (e.g. a variable) must have a defined type.
- Request for an object (the message) must conform operation (method) that the class provides.

Genericity

It should be possible to write classes with formal generic parameters representing arbitrary types.

- It is necessary to have classes that can work with a type that is not known in advance.
- As an example, we can need lists into which we can store objects of different classes (types).

Inheritance and Redefinition

It should be possible to define a class as inheriting from another.

It should be possible to redefine the specification, signature and implementation of an inherited feature.

- Inheritance enables to build a new classes based on existing one. The basic idea is an extension of the original class by new features.
- In the context of inheritance, we can also require changing some features of the original class.

Polymorphism and Dynamic Binding

It should be possible to attach entities (names in the software texts representing run-time objects) to run-time objects of various possible types, under the control of the inheritance-based type system.

Calling a feature on an entity should always trigger the feature corresponding to the type of the attached run-time object, which is not necessarily the same in different executions of the call.

- Sometimes we need a single object featured in a different context in a different role.
- A role means a different behavior, which may vary in time.

Memory Management and Garbage Collection

> The language should make safe automatic memory management possible, and the implementation should provide an automatic memory manager taking care of garbage collection.

- In large programs, many new objects are constructed and destructed over time in different contexts.
- There is a problem to control the life cycle of these object manually.
- The support of object destruction must be done automatically.

Example

Class Declaration

```
#include <iostream>
using namespace std;

const int N = 5;

class KeyValue {
private:
    int key;
    float value;

public:
    KeyValue(int k, float v);
    float GetValue();
};

class KeyValues {
private:
    KeyValue* keyValues[N];
public:
    KeyValues();
    ~KeyValues();
    KeyValue* Item(int indx);
};
```

Using the Class

```
int main() {
    KeyValues* kv;
    kv = new KeyValues();

    for(int i = 0; i < N; i++) {
        cout << kv->Item(i)->GetValue() << endl;
    }

    delete kv;

    getchar();
    return 0;
}
```

Class Implementation (definition)

```
KeyValue::KeyValue(int k, float v) {
    this->key = k;
    this->value = v;
}

float KeyValue::GetValue() {
    return this->value;
}

KeyValues::KeyValues() {
    for(int i = 0; i < N; i++) {
        KeyValue* kv = new KeyValue(i, 10 * i);
        this->keyValues[i] = kv;
    }
}

KeyValues::~KeyValues() {
    for(int i = 0; i < N; i++) {
        delete this->keyValues[i];
    }
}

KeyValue* KeyValues::Item(int indx) {
    return this->keyValues[indx];
}
```

Alternative 1

```cpp
class KeyValues {
private:
    KeyValue* keyValues[N];
public:
    KeyValues();
    ~KeyValues();
    KeyValue Item(int indx);
};
```

```cpp
KeyValue KeyValues::Item(int indx) {
    return *(this->keyValues[indx]);
}
```

```cpp
int main() {
    KeyValues* kv;
    kv = new KeyValues();

    for(int i = 0; i < N; i++) {
        cout << kv->Item(i).GetValue() << endl;
    }

    delete kv;

    getchar();
    return 0;
}
```

Alternative 2

```
class KeyValues {
private:
    KeyValue* keyValues[N];
public:
    KeyValues();
    ~KeyValues();
    KeyValue& Item(int indx);
};
```

```
KeyValue& KeyValues::Item(int indx) {
    return *(this->keyValues[indx]);
}
```

```
int main() {
    KeyValues* kv;
    kv = new KeyValues();

    for(int i = 0; i < N; i++) {
        cout << kv->Item(i).GetValue() << endl;
    }

    delete kv;

    getchar();
    return 0;
}
```

Questions

- What are the main reasons for software changes?
- What are the main factors influencing the object orientation?
- Explain what is the object-oriented method and language.
- Explain what the support of object-oriented implementation.
- Explain what is reusability (using and building libraries).
- Explain the concepts of class and object, and use the correct terminology.
- Explain class properties regarding modularity.
- Explain the principle of encapsulation in OOP.
- Explain the principle of message passing.
- Explain the principles of the declaration and the definition of a simple class in C ++.

Sources

- Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall 1997. [17-36]