Object-Oriented Programming in C++

Grzegorz Jabłoński

Department of Microelectronics and Computer Science (K-25)

Building B18 (next to the library)

gwj@dmcs.p.lodz.pl

Syllabus

- http://neo.dmcs.p.lodz.pl/oopc
  - General overview of C++
  - Classes
  - Fields and methods
  - Operator overload
  - Inheritance
  - Virtual functions
  - Templates
  - Exceptions
  - Class hierarchies
  - C++ standard library (STL)

Lecture Material

- Goals for design
- Design paradigms
- Object-oriented design process
- Object-oriented design basics
  - Abstraction
  - Interfaces
  - Responsibilities
  - Collaborators
- Example
  - Identifying objects
  - Identifying relationships

Software Engineering Goals for Design

- Reusability
  - Develop components that can be reused in many systems portable and independent
  - "plug-and-play" programming (libraries)
- Extensibility
  - Support for external plug-ins (e.g., Photoshop)
- Flexibility
  - Design so that change will be easy when data/features are added
  - Design so that modifications are less likely to break the system
  - Localize effect of changes

Design Process

- Goal: Create a system
- In general, the design process is:
  - Divide/Describe system in terms of components
  - Divide/Describe components in terms of sub-components
- Concept of abstraction
  - Essential for process, hides details of components that are irrelevant to the current design phase
- Component identification is top-down
  - Decompose system into successively smaller, less complex components
- Integration is bottom-up
  - Build target system by combining small components in useful ways
- Design is applied using a paradigm: procedural, modular, object-oriented

Abstraction

- A named collection of attributes and behavior relevant to modeling a given entity for some particular purpose
- Desirable Properties:

Well named	name conveys aspects of the abstraction
Coherent	makes sense
Accurate	contains only attributes modeled entity contains
Minimal	contains only attributes needed for the purpose
Complete	contains all attribute/behavior needed for the purpose

Forms of Abstraction

- Functions (procedural design)
  - Define set of functions to accomplish task
  - Pass information from function to function
  - Results in a hierarchichal organization of functions
- Modules (modular design)
  - Define modules, where each has data and procedures
  - Each module has a public and a private section
  - A module groups related data and/or procedures
  - Works as a scoping mechanism
- Classes/Objects (object-oriented design)
  - Abstract Data Types
  - Divide project in set of cooperating classes
  - Each class has a very specific functionality
  - Classes can be used to create multiple instances of objects

Procedural Paradigm

- Apply procedural decomposition
  - divide the problem into a sequence of simpler sub-problems to be solved independently
- The resulting program will consist of a sequence of procedure calls
- The designer thinks in terms of tasks and sub-tasks, identifying what must be done to whom
- Traditional procedural languages: COBOL, FORTRAN, Pascal, C
- Design notations: structure charts, dataflow diagrams

Problems in Procedural Development

- The result is a large program consisting of many small procedures
- There is no natural hierarchy organizing those procedures
- It is often not clear which procedure does what to what data
- Control of which procedures potentially have access to what data is poor
- These combine to make it difficult to fix bugs, modify and maintain the system
- The natural interdependence of procedures due to data passing (or the use of global data, which is worse) makes it difficult to reuse most procedures in other systems
- High dependence between functions (high coupling)

Procedural Design

- Consider a domain that deals with geometry (shapes, angles, addition of points, etc.)
- A procedural design would have:

```
void distance(int x1, int y2, int x2, int y2, float& distance);
void angle2radian(float degree, float& radian);
void radian2angle(float radian, float& degree);
void circlearea(int centerx, int centery, int radius, float& area);
void squarearea(int x1, int x2, int width, int height, float &area);
void squareperimeter(int x1, int x2, int width, int height, float &prm);
...
```

- Notice that the central aspect of the design is the procedure, and not the data
  - As a matter of fact, there is no data representation

Modular Programming

- This is a relatively simple extension of the purely procedural approach
- Data and related procedures are collected in some construct, call it a module
- The module provides some support for hiding its contents
- In particular, data can only be modified by procedures in the same module
- The design process now emphasizes data over procedures. First identify the data elements that will be necessary and then wrap them in modules
- Typical languages: Ada 83, Modula

Problems with Modular Programming

- Modules do solve most of the (noted) difficulties with procedural programming
- Modules allow only partial information hiding (when compared with OOP)
- Cannot have multiple copies of a module, thus restricting your design solutions

Modular Design

- For the same domain as before, a modular design would have:

```
// Geometry Module
struct Circle { int centerx, centery; int radius; };
struct Square { int x1, x2, width, height; };
Circle *NewCircle(int center, int radius);
Square *NewSquare(int x1, int x2, int width, int height);
float CircleArea(Circle& c);
float SquareArea(Square& s);
float SquarePerimeter(Square& s);
void distance(int x1, int y2, int x2, int y2, float& distance);
void angle2radian(float degree, float& radian);
void radian2angle(float radian, float& degree);
...
```

Notice that the central aspect of the design is still procedure, but there is some data representation

- Also note that the concept of “Point” is not introduced because it is not needed for the design and it does not provide any advantage to have it

Object-Oriented Paradigm

- Think of building the system from parts, similar to constructing a machine
- Each part is an object which has its own attributes and capabilities and interacts with other parts to solve the problem
- Identify classes of objects that can be reused
- Think in terms of objects and their interactions
- At a high level, think of an object as a thing-in-its-own-right, not of the internal structure needed to make the object work
- Typical languages: Smalltalk, C++, Java, Eiffel

Why Object-Oriented?

- First of all, OO is just another paradigm... (and there will be more)
- Any system that can be designed and implemented using OO can also be designed and implemented in a purely procedural manner.
- But, OO makes some things easier
- During high-level design, it is often more natural to think of the problem to be solved in terms of a collection of interacting things (objects) rather than in terms of data and procedures
- OO often makes it easier to understand and forcibly control data access
- Objects promote reusability

OO Design

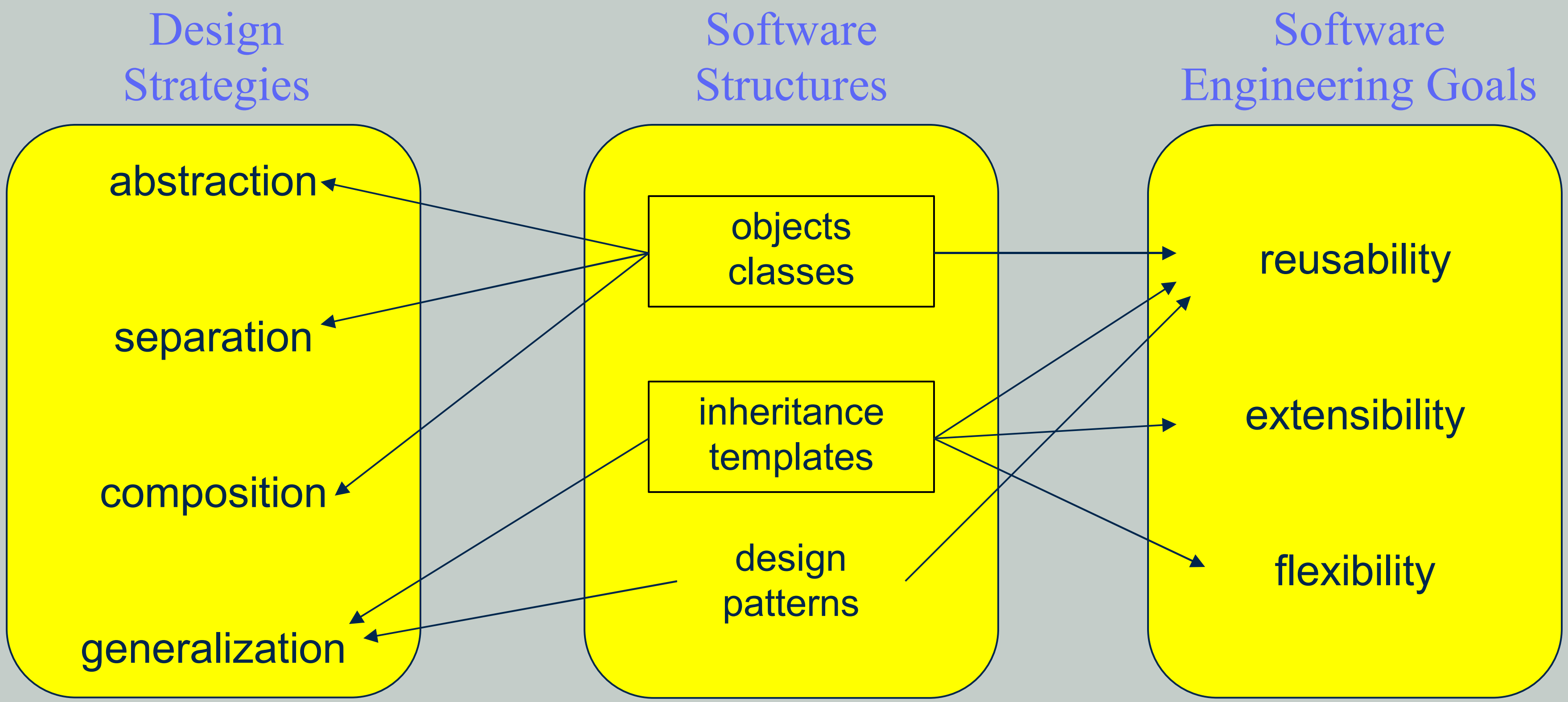
- For the same domain as before, an OO design would have:

```
class Point { ...
  float distance(Point &pt);
};
class Shape { float Area(); float Perimeter(); Point center(); }
class Circle : Shape {
  private: Point center; int radius;
  public: // constructors, assignment operators, etc...
  float Area(); // calc my area
  float Perimeter();
};
class Square : Shape {
  private: Point anchor; int width, height;
  public: // constructors, assignment operators, etc...
  float Area();
  float Perimeter();
};
...
```

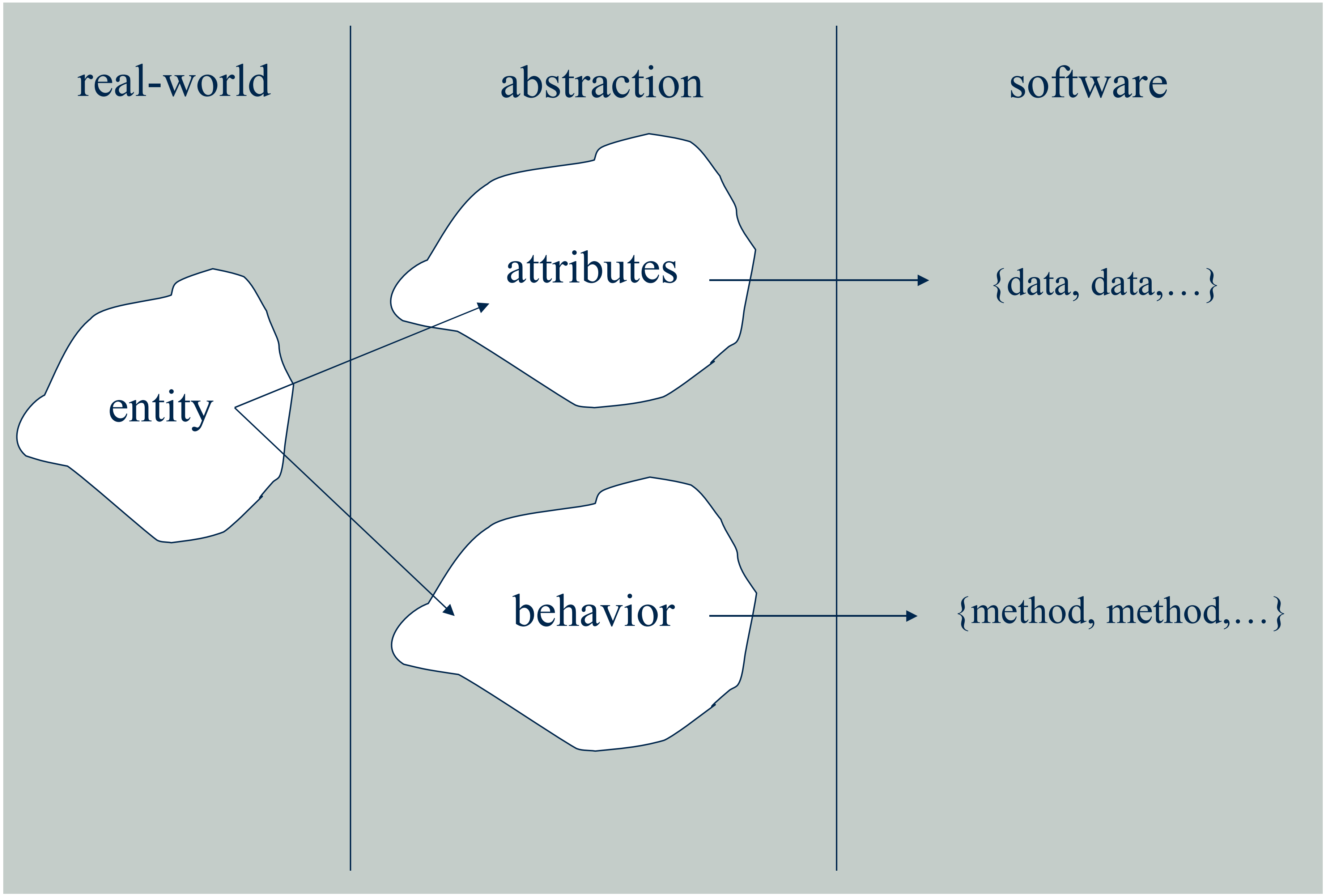
- Notice that the central aspect of the design is now the data, the operations are defined together with the data

Design Strategies in OO Development

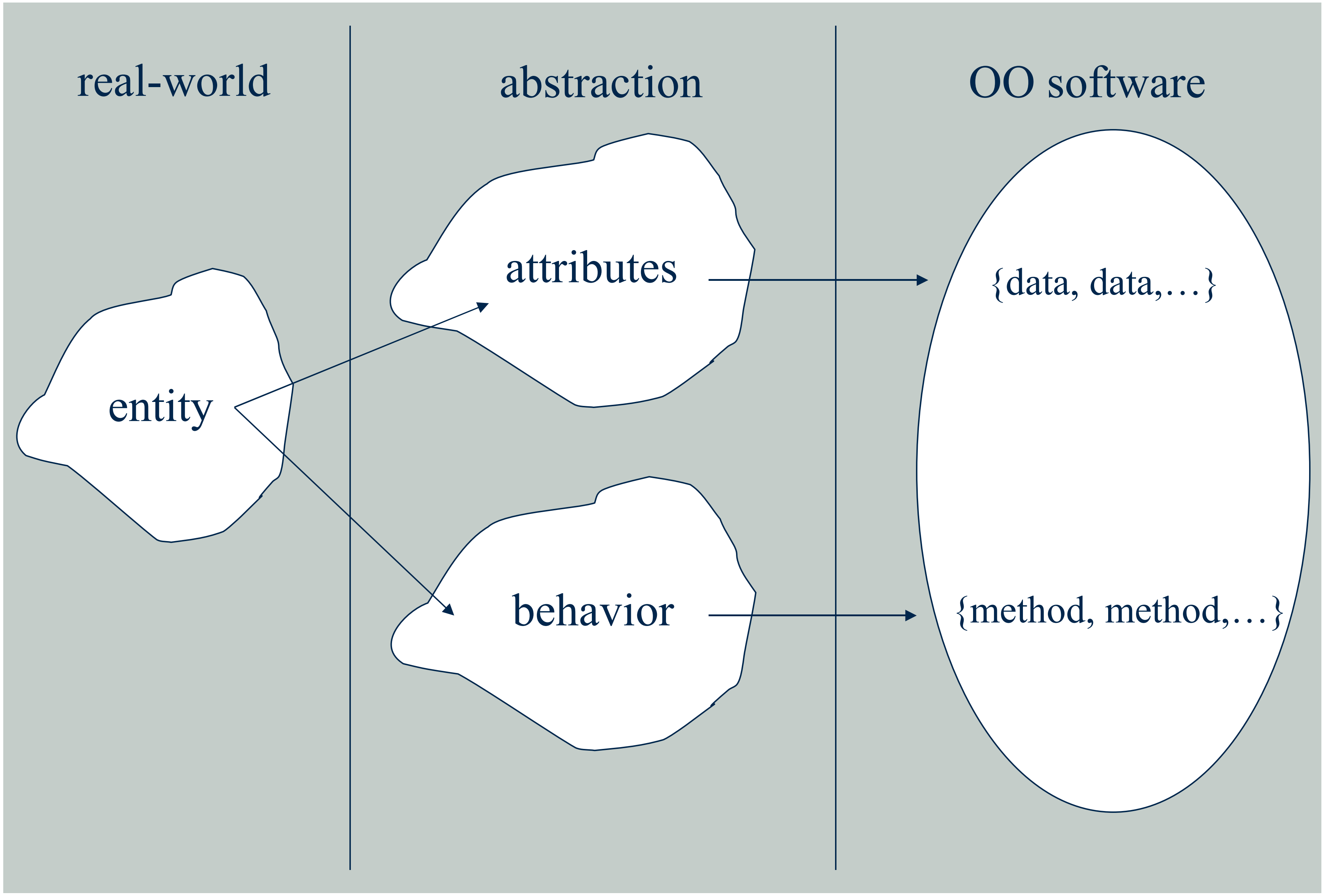
Abstraction	modeling essential properties
Separation	treat *what* and *how* independently
Composition	building complex structures from simpler ones
Generalization	identifying common elements



Mapping Abstraction to Software



Mapping Abstraction to Software in OO



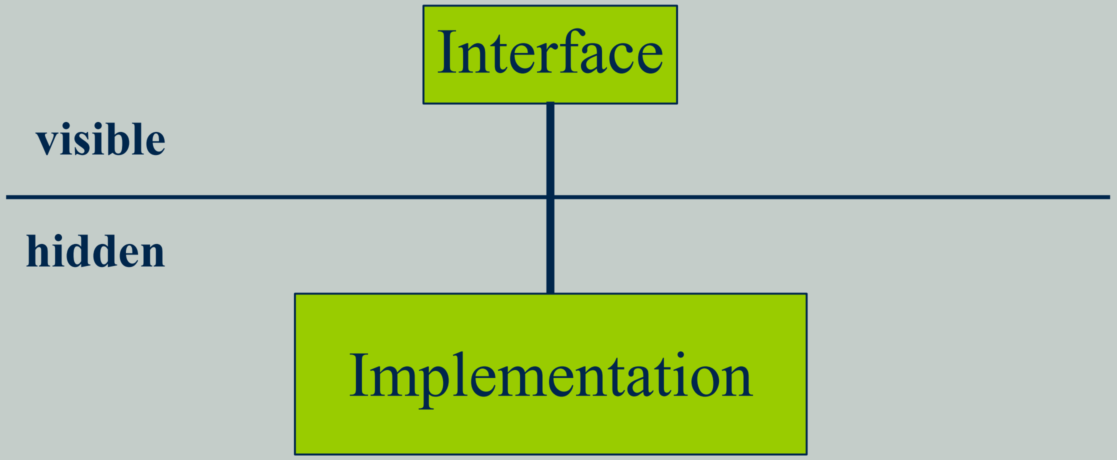
Separation of Interface from Implementation

- In programming, the independent *specification* of an interface and one or more *implementations* of that interface

What is to be done

vs.

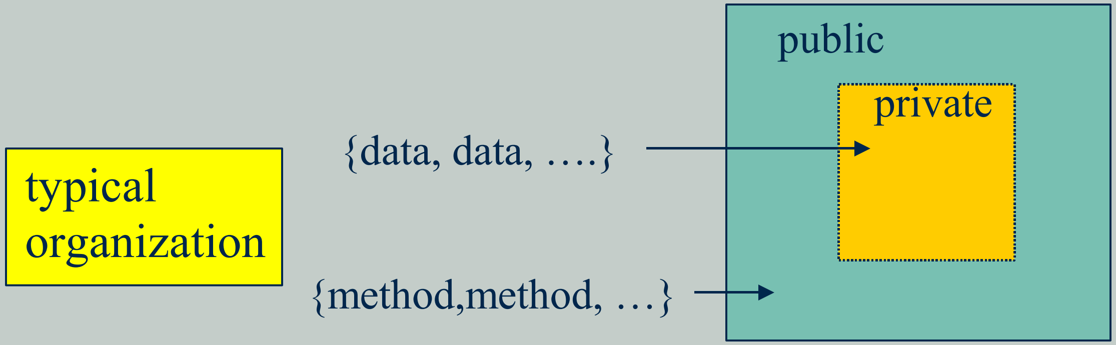
How it is to be done



- Another advantage of this is that we can program depending on the interface without worrying about the implementation
  - Contract-based programming
  - Allows abstraction in the design process

General Structure of a Class

- Class
  - a named software representation for an abstraction that separates the implementation of the representation from the interface of the representation
- A class models an abstraction, which models an entity (possibly "real")
- A class represents all members of a group of objects ("instances" of the class)
- A class provides a public interface and a private implementation
- The hiding of the data and "algorithm" from the user is important. Access restrictions prevent idle, erroneous, or malicious alterations



Object-Oriented Design Process

- Steps to be carried out to design in OO terms
- Restricting the domain: Use Cases (Scenarios, Descriptions of use)
  - Identify **objects** (entities from the domain, data)
  - Identify responsibilities (actions, behavior)
- How to define behavior
  - Identify collaborations
    - Decide whether behavior is accomplised by a single class or through the collaboration of a number of "related" classes
    - Static behavior
      - Behavior always exists
    - Dynamic behavior
      - Depending of when/how a behavior is invoked, it might or might not be legal
  - Identify relationships between objects
    - Composition by association, aggregation, links, others

Getting Started

- In the beginning… there is a specification:

  Specification:

  Design a music catalog system. The system must support adding recordings, storing information about artist, album title, song title, song composer, etc. The user of the system should be able to search the collection for any information. It should also allow the user to browse the collection.

- The specification is usually somewhat unsatisfactory
- Frequently important details are missing
- Often, much is said in the specification that is unimportant

Identifying the Objects

- We must:
  - Identify potential objects from the specification
  - Eliminate phony candidates
  - Determine how the legitimate objects will interact
  - Extrapolate classes from the objects
- This process:
  - Requires experience to do really well
  - Requires guidelines, none of which are entirely adequate
  - Often uses several approaches together
  - Should lead to too many rather than too few potential objects

Several Approaches

- Abbott and Booch suggest:
  - use nouns, pronouns, noun phrases to identify objects and classes
  - singular -> object, plural -> class
  - not all nouns are really going to relate to objects
- Coad and Yourdon suggest:
  - identify individual or group "things" in the system/problem
- Ross suggests common object categories:
  - people
  - places
  - things
  - organizations
  - concepts
  - events

Objects and the Problem Domain

- What constitutes a "potential object" depends on the problem domain
- Discuss with a domain expert — a person who works in the domain in which the system will be used
- Try to identify objects from the way that the users/experts think about the system/problem

Specification:

Design a music catalog system. The system must support adding recordings, storing information about artist, album title, song title, song composer, etc. The user of the system should be able to search the collection for any information. It should also allow the user to browse the collection.

Eliminate "false" Objects

- An object should:
  - be a real-world entity
  - be important to the discussion of the requirements
  - have a crisply defined boundary
  - make sense; i.e., the attributes and behaviors should all be closely related
- Danger signs:
  - class name is a verb
  - class is described as performing something
  - class involves multiple abstractions
  - class is derived from another, but has few features itself
  - class has only one public method
  - class has no methods

Example: Music System

- Looking for nouns
- First cut:
  - music
  - catalog (collection)
  - system
  - user
  - song
  - title
  - artist
  - recordings (album)
  - composer
  - information

Specification:

Design a **music catalog system**. The **system** must support adding **recordings**, storing **information** about **artist**, **album title**, **song title**, **song composer**, etc. The **user** of the **system** should be able to search the **collection** for any **information**. It should also allow the **user** to browse the **collection**.

- **Which of these are important to consider in our design?**
- **Real-world entity, important for problem, clearly definable?**

Example: Music System

- Reject (for now)
  - music (refers to the type of information stored, but nothing musical stored)
  - catalog (collection, system) - all the same
  - information - general name to refer to pieces of the collection
  - user - external to the system, plays a role in the system
  - song, title, artist, composer
  - recordings (album)

- **Do we need to consider "title" a class?**
- **How about a person's name?**

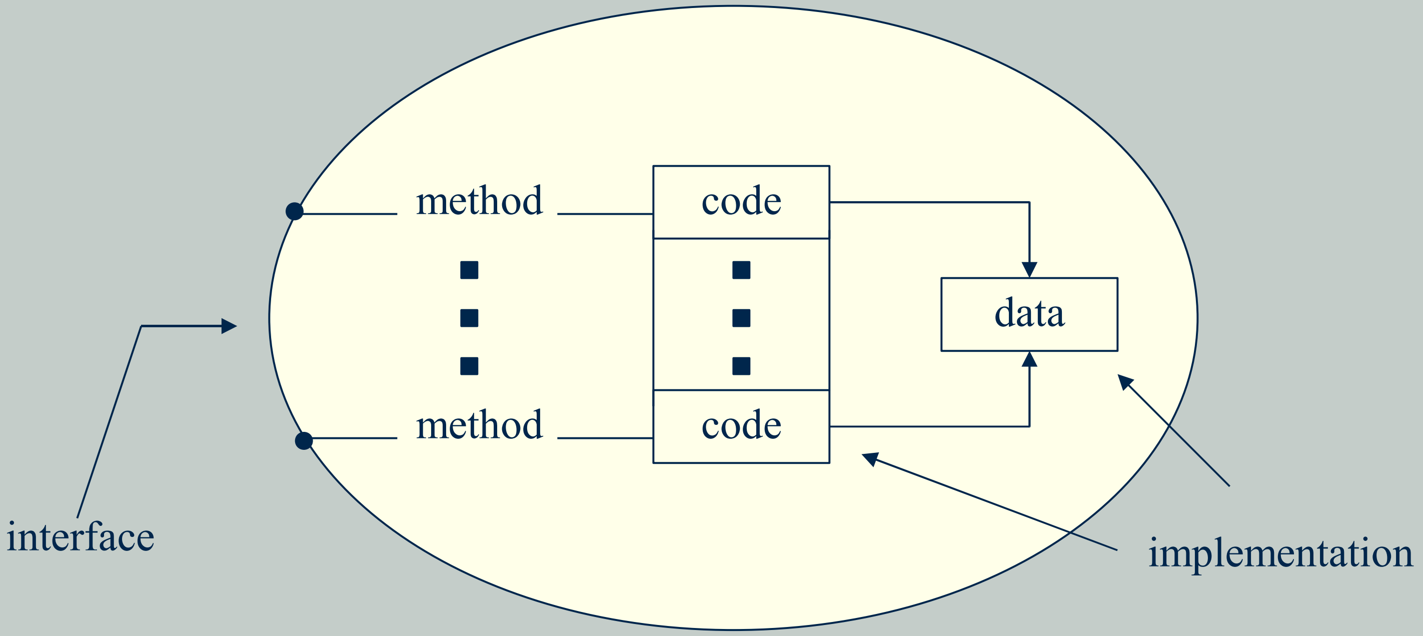
- Data structure support required... (organization of elements in domain)
  - Catalog has a collection of recordings
  - Recording has title, artist name, list of songs
  - Song has title, composer name, artist name

Example: Music System

- What about overall control?
  - The primary controller may be either procedural or an object
  - Collection
    - User uses Catalog, which contains the collection, a list of Recording objects
    - (somehow) provides support for each of the required actions
  - However, the Collection should respond to instructions (events), not seek them out. In the implementation, we must parse an input script, or have a GUI. Either way, that's not part of the Collection, although it would interact with it

General Structure of an Object

- Object:
  - a distinct instance of a given class that encapsulates its implementation details and is structurally identical to all other instances of that class
- An object "encapsulates" its data and the operations that may be performed on that data
- An object's private data may **only** be accessed via the member functions defined within the object's class
- An object hides details of representation and implementation from the user


method
code
data
method
code
interface
implementation