

## Design 3

- Heuristics.
- Performance and optimization.
- Persistence.

1

## Heuristics

- A heuristic is a rule of thumb. Heuristics serve as guidelines.
- Heuristics represent good practices that have been learned through experience.
- Design heuristics promote better quality of designs.
- Many design heuristics apply to all design approaches, while some only pertain to object-oriented designs.
- A good reference is the book by Arthur J. Riel, *Object-Oriented Design Heuristics*, Addison-Wesley, 1996.

2

## Some Software Engineering Principles

- Modularity – divide and conquer.
- Abstraction – creating the illusion.
- Encapsulation – hiding the implementation.
- Coupling – simplifying the connections.
- Cohesion – providing a single abstraction.

3

## Some Questions (1 of 2)

- What should be the visibility of attributes of an object?
- What about accessor methods?
  - When do we need them?
  - What should their visibility be?
- What about the visibility of other operations?
  - Which should be public? Private? Protected? Package?

4

## Some Questions (2 of 2)

- What about the number of attributes?
- What about the number of operations?
- How do you recognize when a class is too complicated and can be split?
- How do you split a complicated class?

5

## The God Class Problem (1 of 2)

- In Booch's book on OO Design, he uses a home heating system as an example.
- There are two sensors in each room, one to determine the actual temperature, and one to determine whether there is anyone in the room.
- There is also a thermostat that indicates the desired temperature.

6

## The God Class Problem (2 of 2)

- Business rules:
  - If there is no one in the room, the temperature is maintained at five degrees below the thermostat setting.
  - But if someone is in the room, the temperature is brought up to the thermostat setting.
- How should this be modeled?
  - What are the classes?
  - How is information encapsulated?
  - How do we avoid the "god class" problem?

7

## Reducing Coupling and Increasing Cohesion

- Invent aggregator classes (e.g., Room in the home heating example).
  - Aggregators should be aware of the parts they contain, but the parts need not be aware of the aggregator.
- Objects should not allow their implementations to leak through the interface (encapsulation).
- Objects should embody a single abstraction.
  - Split complicated objects into simpler ones.

8

## Performance

- Determine where the performance bottlenecks are.
  - First make it right. Then make it fast.
- Use the right data structures and algorithms.
- Shorten communication paths.
  - Object folding.
  - Short circuiting references.

9

## Object Folding (1 of 2)

- If two objects need to communicate a lot, they are tightly coupled.
- A time performance advantage might be gained if the two objects were combined.
  - The operations that were part of one object would now have direct visibility to the data that was owned by the other object.

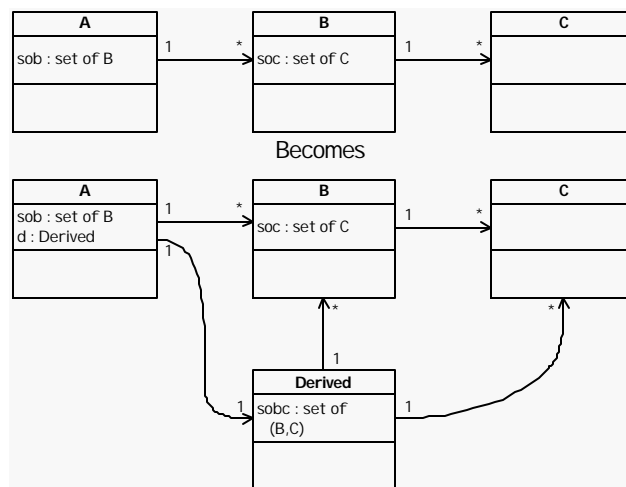
10

## Object Folding (2 of 2)

- Works best if there is 1:1 multiplicity relationship between the two objects.
- If there is 1:\* multiplicity:
  - Copy the information from the 1 object into each instance of the \* object.
  - This duplication of information violates *first normal form* of data modeling.

11

## Short Circuit References



12

## Caching Derived Data (1 of 2)

- Derived attributes – computed each time they are needed.
- Once you compute the derived attribute, it may continue to be valid for some amount of time.
- You can avoid re-computing it if you cache its value.

13

## Caching Derived Data (2 of 2)

- Tradeoffs:
  - Space/Time – Does the derived data take up a lot of space? Does it take a long time to re-compute?
  - Volatility – How frequently does the underlying data change? What is the cost of checking for validity?
  - Frequency of access – How often do we need the derived data? Is it worth keeping?

14

## Object Lifetimes (1 of 2)

- In some languages (e.g., Eiffel, Ada, Smalltalk, Java), when the number of variables that point to an object falls to zero, the space taken up by the object is automatically reclaimed by the *garbage collector*.
- In other languages (e.g., C++), it is the responsibility of the programmer to reclaim the space before removing the last reference to an object. If this does not happen, a *memory leak* will result.

15

## Object Lifetimes (2 of 2)

- Destroying an object that has references pointing to it (in C++) will result in an addressing exception (or worse) if any of these references are attempted.
- At design time, if we know the target language will be C++, we may want to design additional code to determine when it is necessary and safe to destroy objects.

16

## Persistence (1 of 8)

- We need to use a persistence mechanism to preserve the state of an object in any of the following situations:
  - The program is not continuously running, and we need to restore the last value of an object the next time the program is started.
  - Multiple programs share the same objects.
  - The system needs to be robust (so that data is not lost if there is a hardware or software failure).

17

## Persistence (2 of 8)

- A persistence mechanism may be:
  - A database management system (DBMS). It may be hierarchical, relational or object-oriented.
    - Advantages: the DBMS manages the storage, checkpoint, restore, locking, security, integrity, etc. of the data.
    - Disadvantages: performance may be poor.
  - A flat file.
    - Advantages: under programmer control (performance tends to be better).
    - Disadvantages: programmer is responsible for managing the storage, checkpoint, restore, locking, security, integrity, etc. of the data.

18

## Persistence (3 of 8)

- Levels of persistence:
  - Transient data computed during the evaluation of an expression.
  - Local variables in blocks of code.
  - Local variables within methods.
  - Instance attributes.
  - Class attributes.
  - Data that persists between executions of a program.
  - Data that outlives the program, or is used by multiple programs (enterprise data).

19

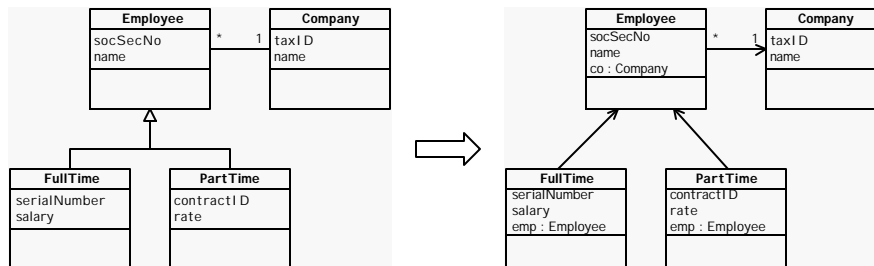
## Persistence (4 of 8)

- Mapping an OO model to a relational model:
  - The relational model only shows data.
  - No inheritance, no classes, no operations.
- Classes → Tables.
- Object instances → Rows.
- Attributes → Columns.
- References to other objects → Foreign keys.

20

## Persistence (5 of 8)

- Simulate inheritance through delegation.



21

## Persistence (6 of 8)

- What about class (static) attributes?
  - Two choices:
    - Store in each row of the table.
    - Create a separate table for class attributes.
      - Problem: each row probably has a different data type.

22

## Persistence (7 of 8)

- When we re-hydrate an object, the constructor should retrieve the value from the DBMS.
- What should we do when we send a message to that object (and change its attribute values)? Possibilities:
  - Use the DBMS for initialization only.
  - Write to the DBMS when the data changes.
  - Write to the DBMS when the program terminates, or at other checkpoints.

23

## Persistence (8 of 8)

- When the object data in the DBMS changes, should we keep the replicate objects in sync with the DBMS? Two possibilities:
  - Yes, always.
  - No, use the DBMS for initialization only.
- How do we keep the replicate objects in sync with the DBMS?
  - Push strategy.
  - Pull strategy.

24