Topic 8

1. Object oriented programming
2. Implementing a simple class
3. Specifying the public interface
4. Designing the data representation
5. Member functions
6. Constructors
7. Problem solving: tracing objects
8. Problem solving: discovering classes
9. Separate compilation
10. Pointers to objects
11. Problem solving: patterns for object data

Discovering Classes

When you solve a problem using classes, you need to determine the classes required based on the English (or other informal) problem description.

You may be able to reuse existing classes, or you may need to implement new ones.

To discover classes and member functions, look for *the:*

1. ***Nouns*** in the problem description: at least one of them will be a class. Classes almost always represent things (data), not actions.
2. ***Verbs***: they will be the member functions. Functions do things to or with the data

Discovering Classes: Concepts

Generally, concepts from the problem domain,
be it science, business, or a game, make good classes.

The name for such a class should be
a noun that describes the concept.

Other frequently used classes represent
system services such as files or menus.

Not Discovering Classes

What might *not* be a good class?

If you can’t tell from the class name what
an object of the class is supposed to do,
then you are probably not on the right track.

Classes: Bad/Good Example

For example, you might be asked to write a program that prints paychecks.

You start by trying to design a `class PaycheckProgram`.

What would an object of this class do? Lots, too much.

A better class would be:

`class Paycheck`

Paycheck Class

You can *visualize* a paycheck object.

You can then think about useful member functions of the **`Paycheck`** class, such as **`compute_taxes`**, that help you solve the problem.

“Has-a” relationship

When you analyze a problem description,
you often find that you need multiple classes.

One of the fundamental relationships between classes
is the “*aggregation*” relationship

(which is informally known as the “has-a” relationship).

Consider a quiz that is made up of questions.

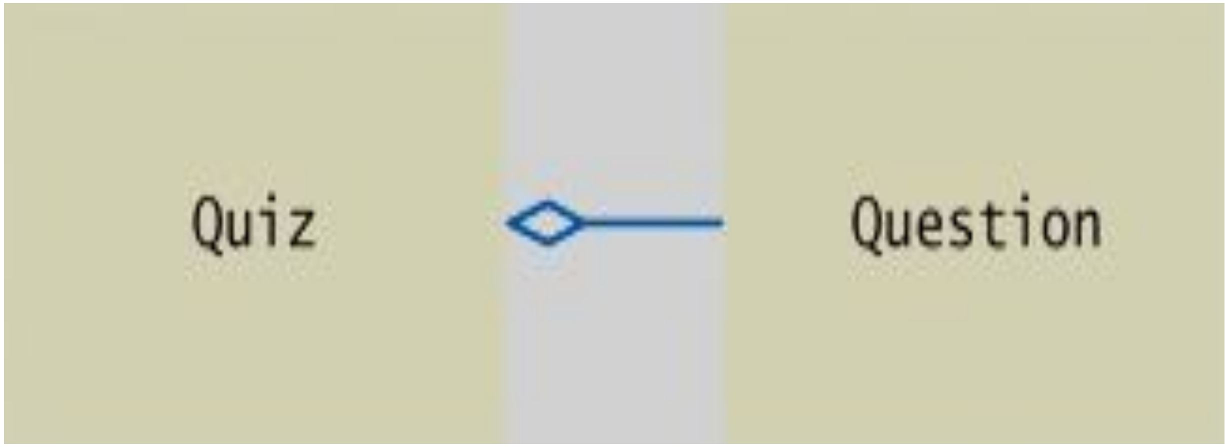
Since each quiz has one or more questions,
we say that the class **`Quiz`** *aggregates* the class **`Question`**

UML (Unified Modeling Language)

There is a standard notation to describe class relationships:

(Unified Modeling Language)

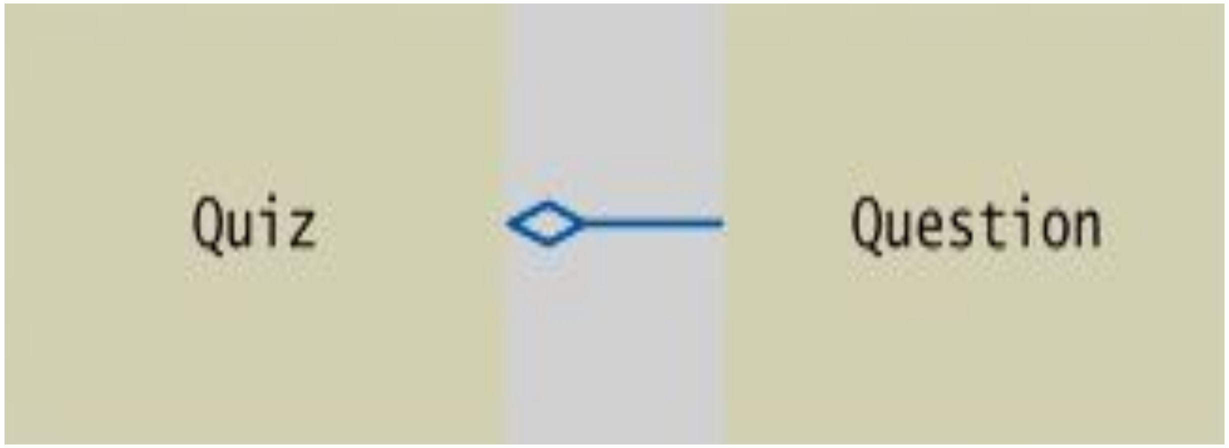
In the UML notation,
aggregation is denoted by a line
with a diamond-shaped symbol



UML Example: Quiz of Questions

The problem states that the **`Quiz`** object manages lots of **`Question`** objects.

The code follows directly, using a vector to mange the **`Question`**s:



```
class Quiz
{
...
private:
vector<Question> questions;
};
```

Discovering Classes: Review

In summary, when you analyze a problem description, you will want to carry out these tasks:

- Find the concepts that you need to implement as classes. Often, these will be nouns in the problem description.
- Find the responsibilities of the classes to encode as functions. Often, these will be verbs in the problem description.
- Find relationships between the classes that you have discovered.

Practice It: Discovering Classes

1. Suppose you are to design a software system to maintain the circulation system of a library. It has a list of library patrons and information on all items that can be borrowed by patrons, such as books, CDs, and DVDs. What are some classes you might create?

2. Suppose you are to design a software to maintain a collection of recipes. Each recipe will contain a list of ingredients as well as step-by-step instructions. Users should be able to find recipes that contain certain ingredients. What are some classes you might create?

Programming Tip 9.3
Make Parallel Vectors into Vectors of Objects

- You might use vectors of the same length, each of which stores a part of what conceptually should be an object.
  - Instead, reorganize your program and use a single vector whose elements are objects.

- For example, suppose an invoice contains a series of item descriptions and prices. You could keep two vectors:
  - `vector<string> descriptions;`
  - `vector<double> prices;`

- Parallel vectors become a headache to ensure that the vectors always have the same length and that each slice is filled with values that actually belong together

Make Parallel Vectors into Vectors of Objects

- Instead of:
  - `vector<string> descriptions;`
  - `vector<double> prices;`

- Make an Item class:

```
class Item
{
public:
    . . .
private:
    string description;
    double price;
};
```

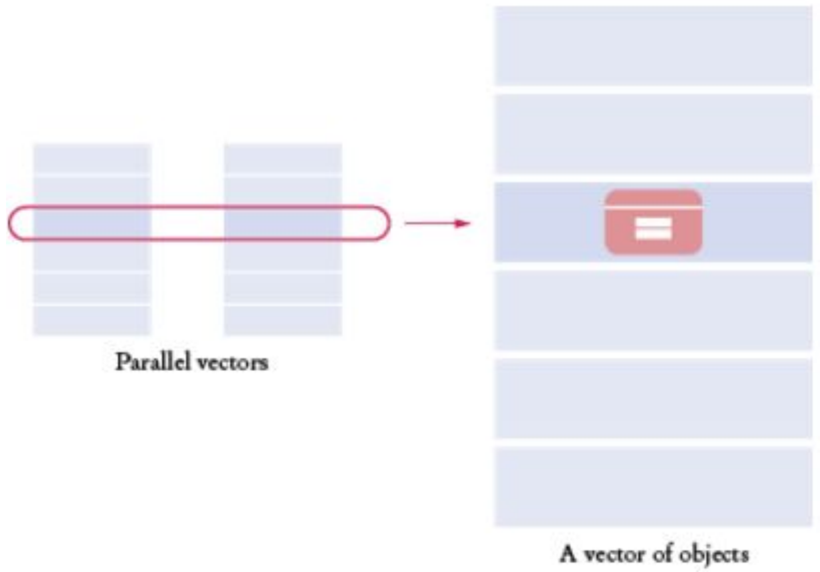


Figure 8 Eliminating Parallel Vectors

- Replace the parallel vectors with a single vector:

```
vector<Item> items;
```