CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 18
April 5, 2016

Demo: Hangman Game

- Game Rules
- Code Design
- Storage Management
- Refactored Game

Demo: Hangman Game

Game Rules

Hangman game

Well-known letter-guessing game.

Start with a hidden *puzzle word*.

Player guesses a letter.

- If letter appears in puzzle word, matching letters are uncovered.
- If letter does not appear, it is shown in list of bad guesses.

Player wins when puzzle word is uncovered.

Player loses after 7 bad guesses

Code Design

Overall design

Game elements:

1. Puzzle word and letters found so far.
2. Bad guesses word.
3. Alphabet and letters left.
4. Vocabulary.
5. Game board display (viewer).
6. Game play (controller).

Use cases

Two levels.

1. Play one round of Hangman on a puzzle word.
   - Get input letter from user.
   - Classify input as good, bad, redundant, or not allowed.
   - Inform user and show updated board.
   - Announce termination and win/loss.
2. Repeated play
   - Choose unused word from vocabulary.
   - Play Hangman with that word.
   - Tally and announce win/loss.
   - Ask user whether to play again.

Code structure: Model

Model

1. `Alphabet` used to represent letters left.
2. `HangWord` used to represent puzzle word and bad guesses.
3. Both are derived from `BaseWord`
4. Common elements are a word and a visibility mask.
5. Variable elements:
   - How to print masked word.
   - Operations needed: `find` and `try`
6. Class `Board` data members store model state.

Code structure: Viewer and controller

Viewer Contained in class `Board`.

- `Board::print()` prints the puzzle, letters left, and bad guesses.
- `Board::move()` prints guess, outcome, and next board.
- `Board::play()` prints the win/loss message.

Controller Contained in class `Board`.

- `Board::play()` carries out turns and determines game termination.
- `Board::move()` prompts users for character and carries out turn.
- `Board::guess()` updates the model.

Class `Game`

Class `Game` is a top-level MVC design.

- **Model** contains alphabet, remaining vocabulary, and win/loss counters.
- **Viewer** is embedded in `Game::play()`.
- **Controller** is in `Game::playRound()` and `Game::play()`.

Storage Management

Storage management

Two storage management issues in Hangman:

1. How to store the vocabulary list?
2. How to store the words in the vocabulary?

Natural solutions are to store vocabulary as an array of pointers to strings.

Natural way to each string is to use `new` to allocate a character buffer of the appropriate length.

Design issues:

- How big should the vocabulary array be?
- Who owns the strings and takes responsibility for cleanup when they are no longer needed?

String store

A `StringStore` provide an alternative way to store words.

Instead of using `new` once for each string, allocate a big `char` array and copy strings into it.

When no longer needed, `~StringStore()` deletes entire array.

Advantages and disadvantages:

- *Much* more efficient—(each `new` consumes minimum of 32 bytes on modern machines).
- Simpler storage management—ownership of storage remains with `StringStore`.
- Downside: Can't reclaim storage from individual strings until the end.
- How big should the `char` array be?

Refactored Game

Refactored hangman game

Demo `18b-Hangman-full` extends `18a-Hangman` in three respects:

1. It removes the fixed limitation on the vocabulary size.
2. It removes the fixed limitation on the string store size.
3. It more clearly separates the model of `Board` from the viewer/controller.

We'll examine each of these in detail.

Flex arrays

A `FlexArray` is a growable array of elements of type `T`.

Whenever the array is full, private method `grow()` is called to increase the storage allocation.

`grow()` allocates a new array of double the size of the original and copies the data from the original into it (using `memcpy()`).

Note: After `grow()`, array is 1/2 full.

By doubling the size, the amortized time is $O(n)$ for n items.

Flex array implementation issues

Element type: A general-purpose `FlexArray` should allow arrays of arbitrary element type `T`.

If only one type is needed, we can instantiate `T` using `typedef`. Example: `typedef int T;` defines `T` as synonym for `int`.

C++ templates allow for multiple instantiations.

Class types: If `T` is a class type, then its default constructor and destructor are called whenever the array grows.

They must both be designed so that this does not violate the intended semantics.

This problem does not occur with numeric or pointer flexarrays.

String store limitation

Can't use `FlexArray` to implement `StringStore` since pointers to strings would change after `grow()`.

Instead, when one `StringStore` fills up, start another.

Only really want another *storage pool*, not another `StringStore` object.

Eacn new `Pool` is linked to the previous one, enabling all pools to be deleted by `~StringStore()`.

Refactoring `Board` class

Old design for `Board` contained the board model, the board display functions, and the user-interaction code.

New design puts all user interaction into a derived class `Player`.

This makes a clean separation between the *model* (`Board`) and the *controller* (`Player`).

The *viewer* functionality is still distributed between the two.

What are the pros and cons of this distribution?