Lecture 12: Control Styles

Kenneth M. Anderson

Object-Oriented Analysis and Design

CSCI 6448 - Spring Semester, 2005

Goals for this Lecture

- Last Lecture
  - Designing Collaborations
- This Lecture
  - Designing Control Styles
  - Or, designing the decision making processes of your application

Application Control and Control Style

- How does an application respond to events? How does it make decisions?
  - Typically via “control centers”; groups of objects that are in charge of the decision making process
- Control style affects how intelligence is distributed among objects within a control center
  - A control style can be centralized, delegated, dispersed, or somewhere in between

Centralized Control

- A centralized control style places major decision making responsibilities in a small number of objects; those stereotyped as controllers
  - Most objects used by controllers are devoid of any significant decision making responsibilities
    - they are told what to do and they do it!
- A variation of this style is the clustered control style; here decision-making responsibilities are assigned to several controllers, each working on a small part of the overall control
  - Typically there is then one controller which “controls” each cluster; “one controller to rule them all and in the darkness bind them!” (obligatory Lord of the Rings reference!)

Problems

- With centralized control, generally one object (the controller) makes most of the important decisions
  - This is good since it centralizes control logic, but…
- several problems can occur including
  - Control logic can get overly complex
  - All other classes may become information holders
    - this means that most responsibilities move to the controller which then loses cohesion
  - Controllers can become dependent on the contents of their information holders
    - If the information changes, the controller has to change; if there are too many information holders, the controller becomes highly coupled

Delegated Control

- In a delegated control style, the designer makes a concerted effort to delegate decisions, not only between controllers, but also to objects that have other responsibilities
  - Decisions made by controllers are limited to deciding what should be done; other objects then perform that task

Advantages

- Delegated control is more “object oriented” and leads to several benefits
  - Delegating coordinators tend to know about fewer objects than dominating controllers
    - This leads to a loosely coupled system
    - Changes typically affect fewer objects
  - Dialogs are higher-level
    - Collaborations between coordinators and the objects they coordinate tend to be higher level requests rather than simple requests to store and retrieve data
      - e.g. calculateTaxes() versus getTaxRate()
  - It is easier to divide design work among team members
    - More objects with interesting responsibilities makes it easy to divide design and implementation work among the development team

Problems

- Too much distribution of responsibility can lead to weak objects and weak collaborations
  - Carried to extremes, a delegated control style can result in objects that do not “know” or “do” enough to be interesting
- Look for these warning signs
  - Small service providers used by a single client; the service was factored out of a controller and should be merged back in
  - Complicated collaborations between delegator and delegates; this can happen when not enough context is passed with a request
  - Lots of collaborations but not much work getting done

Dispersed Control

- A dispersed control style distributes decision-making responsibilities across many objects involved in a task
  - Benefits
    - Decision making logic becomes very simple
  - Problems
    - May become hard to identify where a particular decision is being made in a system

Developing Control Centers

- In all but the simplest of applications, you will have multiple control centers to design
- Control design is important when controlling
  - user-initiated events
  - complex processes
  - work within a specific object neighborhood
  - external software systems
- In each of these situations
  - pick a control style, and work on specific responsibilities and collaborations

Mixing Styles

- However, do not try to use the same control style everywhere
  - Develop a control style suited to each control situation; ask these questions
    - How are decisions made in this situation?
    - Who should make them?
    - What decisions should be delegated?
    - What patterns of delegation should be established and repeated
- It is best to design collaborations so similar things work similarly
  - For instance, use cases that handle the same kind of user interactions should use the same control style even if the participating objects are different

Example: Speak For Me

- Imagine a software system designed to help a severely disabled user, one who is paralyzed, blind, and cannot speak
  - All this person can do is blink their eyes to indicate “yes” and “no”
- This system allows this user to compose and send messages by speaking the alphabet and allowing the user to select letters to form into words and words into sentences
  - The user can indicate words by selecting a “space”, which is presented after the user has selected at least one letter
  - Several two letter words are used as commands: ES for “end sentence”, SM for “send message”, etc.
- This system is similar to software used by Stephen Hawking, the famous physicist, although he can see and can move his fingers

Example: Build a Message

- We are going to design a control center for Speak For Me that manages the process of building a message
  - Speak For Me speaks letters until one is selected; when a letter is selected, it is spoken and then added to the current word
  - Based on the current message, Speak For Me can try to guess the user's intentions
    - e.g. it can make guesses at the word that the user is trying to spell; if so it speaks the words and allows the user to select the correct word
    - it can also make guesses at the sentence that the user is trying to speak; e.g. it stores all previous sentences composed by the user for re-use; if so, it speaks sentences and allows the user to select the correct sentence

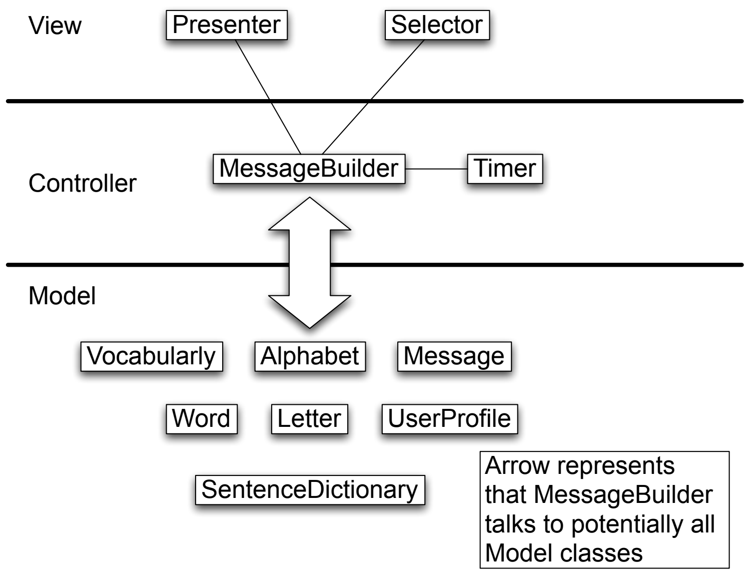
Example: Actions and Responsibilities

- When composing a message:
  - If letter selected, speak letter, add letter to current word
  - If space selected, speak space, add word to end of current sentence, start new word
  - If word selected, speak word, add to end of current sentence, start new word
  - If sentence selected, speak sentence, add sentence to message, start new sentence with new word
- Repeat until a command is issued
  - Processing a command is a separate use case

Example: MessageBuilder

- Candidate: Message Builder
- Purpose: The MessageBuilder is a hub of activity in the application. It coordinates the timing, the presentation of guesses, and the message construction. It centralizes control and is a core element of the control architecture
- Stereotype: Controller or Coordinator?

Example: Architecture



Example: Control Strategy

- When Timer “ticks” MessageBuilder presents its next “guess” via the Presenter
  - e.g. based on the current message, it may decide that the user is trying to spell the word “Chicago”; if so, it will have the presenter speak this word
- When a selection comes in, MessageBuilder will process it
  - Adding letters to word, words to sentences, sentences to the message, or executing commands as needed

Example: Initial Implementation

- See the initial attempt at coding the MessageBuilder, using a centralized control style, in the mb1 directory of the example source code
  - All source code is available from the class website
- All decision logic is placed in the MessageBuilder; other objects have very simple responsibilities
  - A lot of code was not shown; it would consist of various additional complex if statements to handle all of the various states

Discussion

- While all control logic is centralized; the number of states is causing the code to be very complex, with lots of conditionals, use of boolean flags, and the like
- What we want is a way to make the MessageBuilder alter its behavior based on its current state
  - State Pattern to the rescue!
  - It is designed for exactly this context!

State Pattern (I)

- Problem: How to design an object to alter its behavior based on internal state changes
- Context: Sometimes you need to make complex decisions about what to do based on the current state of an object. An object's state can be represented by a number of different objects; The object must change its behavior based on the "current state"

State Pattern (II)

- Forces: Complex, multipart conditional expressions are often used to decide how to proceed; but this can result in code that is hard to maintain
- Solution: Instead of writing code that specifically checks what state an object is in before deciding how to react, design one new class for each possible state that the object can be in. Reassign responsibilities for handling events to each state object; delegate all responsibilities to the state objects and pass in whatever context is needed for them to do their work

Example: States needed for MessageBuilder

- Idling - not doing anything
- Guessing Letters Only - new word has started
- Guessing Letters and Space - at least one letter has been added to current word, so add “space” as an option
- Guessing Letters, Words, and Sentences - at least two letters have been added to current word
- Ending Word - space or word has been selected; check to see if word is command
- Execute Command - command detected
- Suspended - allows user to “pause” message building

Example: Code for Three States

- See state-based implementation in the mb2 directory of the example source code
  - This is an example of a clustered control style; MessageBuilder has delegated decision making logic to each individual state
- These three states contain simpler logic but handle everything that was handled in the original code example
  - Be sure to look at the simplified implementation of MessageBuilder

Example: Switching Styles

- Even though we make use of the State design pattern, we are still using a centralized control style
  - And we mentioned that a delegated control style was more “object oriented”; since it typically leads to a situation where intelligence and responsibilities are more evenly distributed
- So, lets see how we might use a delegated control style in Speak For Me

Example: Making "Letter" Smarter

- Currently, our state keeps track of what the current selection is;
  - GuessingLettersOnly for instance knows that the current selection is a Letter and so it can just directly add it to the message, without checking its type
  - But, why not shift the responsibility of adding the selection to the message to the selection itself
    - after all each object is aware of its own identity; if the selection is a Letter, it knows that it has to call "addLetter() to add itself to the Message object

First, define a new role

- Since SpeakForMe will eventually make "guesses" about words and sentences, we will define a role called Guess
  - A Guess is responsible for knowing how to present itself and knowing how to add itself to a message
- We will make Letter a subclass of Guess and eventually we'll define classes called Word and Sentence that will also be subclasses of Guess; (we could also make Guess an interface)

Second, simplify MessageBuilder

- Since each selection (a Guess) knows how to add itself to a message, the code for handleSelection() in MessageBuilder becomes
  - public void handleSelection() {
    - selection.addTo(message);
  - }
- We've completely delegated the responsibility of adding the selection to the current message to the selections themselves

Delegating Guessing

- The rest of the logic in the old version of MessageBuilder dealt with coming up with a guess
  - we only showed code for guessing the next letter…eventually we would have to add code that would try to guess the word a user was trying to spell, or code that would try to guess the sentence a user was trying to create
- We want to have some other object do the guessing, that way all MessageBuilder has to do is ask for the next guess and present it

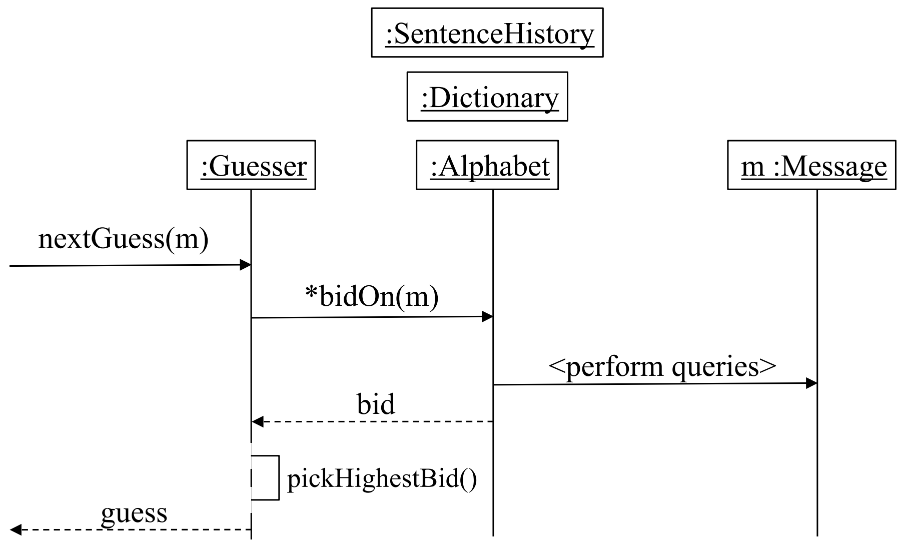
Blackboard Pattern (I)

- To take care of guessing, we will make use of another pattern, called Blackboard
  - In Blackboard you have four roles
    - A blackboard object that stores a particular message
    - A knowledge source that looks at the current message and makes a "bid" about how the current message should be modified
    - A bid that stores information about proposed modifications and the value associated with each of them
    - A controller that asks the knowledge sources to make bids on the current blackboard and then selects one of the bids based on some sort of evaluation process
      - We'll use "highest bidder wins" :-)

Blackboard Pattern (II)

- In SpeakForMe, we will have the following objects play these roles
  - Guesser - plays the role of the controller; MessageBuilder delegates the "guessing responsibility" to this object
  - Alphabet - plays the role of a knowledge source; it has to come up with a letter to present to the user based on the current value of the message
    - eventually we can add additional knowledge sources, such as a Dictionary to guess words, and a SentenceHistory to guess sentences, etc.
  - Message - plays the role of the Blackboard
  - Bid - plays the role of the Bid (pretty original!)

Example: Guesser Architecture



Example: Simplified MessageBuilder

- With this new collaboration, we can eliminate all of the state objects from the clustered implementation of MessageBuilder
- handleTimeout becomes

```
public void handleTimeout() {
    selection = guesser.nextGuess(message);
    selection.presentTo(presenter)
}
```

- MessageBuilder is no longer a controller; it is simply a coordinator; all decision logic has been delegated to other classes!

Discussion and Summary

- Advantages of Delegated Style
  - Control architecture stays the same in the presence of a new “knowledge source”
    - Simply add new type of Guess and a new KnowledgeSource; nothing else changes
  - In a centralized control style:
    - the logic of the controller would change to become aware of the new type of Guess and the conditional logic of the knowledge source
  - In the clustered style
    - a new state would encapsulate this knowledge, but the other states would have to change to take advantage of the new state transitions