Dealing with Change

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/6448 — Lecture 7 — 09/16/2008

Lecture Goals

- Review material from Chapter 3 of the OO A&D textbook
  - Dealing with Change
  - More on Requirements and Use Cases
  - Use Case Styles
  - Discuss the Chapter 3 Example: Todd & Gina's Dog Door, Take 2
  - Emphasize the OO concepts and techniques encountered in Chapter 3

Things Change...

- The one constant in software analysis and design is CHANGE
  - This is true because that's the one constant we face in life
- In software development, requirements always change!
  - No matter how well you design an application, things will change for you:
    - new techniques, new tools, new solutions
  - and things will change for your user:
    - new requirements, new ideas, new needs
- Rather than fight it, you need to:
  - Plan for **likely** change and design your software to accommodate it
  - Document your current state with clear requirements and good use cases
    - When change comes, you'll be able to identify exactly what has changed and where

Todd and Gina's Dog Door

- With respect to the example
  - back in chapter 2, Todd and Gina LOVED the system you designed
  - BUT… the real world intrudes!
    - They are tired of having to listen for Fido! They sometimes miss his barking and he "takes care of business" inside!
    - Also, they are constantly losing the remote!
- So, they have a GREAT idea
  - What if the dog door opened automatically when Fido barked at it?

What's the Process?

- As software engineers, we would like to have a process that we follow
  - So, how do we deal with change?
- In OO A&D, the answer typically is
  - find the use case that most closely matches the change request
  - update the use case to document the new scenario
    - customer focus: IF the system were changed to handle the new request, how would the user interact with it?
  - consider alternate paths (if needed)
  - update the requirements list (use use cases to validate completeness)

Initial Idea

- To allow the dog door to open automatically, we will assume the existence of a “bark recognizer”
  - we won’t try to specify an implementation at this point
    - that might over-constrain our subsequent analysis and design work
  - but we need to introduce some new element to the system to enable the redesign of the use case
- Now, lets examine how the use case changes… this will give us information on how our system’s behavior changes
  - and that will provide insight into how the implementation will need to change

Current Use Case

What the Door Does

1. Fido barks to be let out.
2. Todd or Gina hears Fido barking.
3. Todd or Gina presses the button on the remote control.
4. The dog door opens.
5. Fido goes outside.
6. Fido does his business.
   6.1 The door shuts automatically
   6.2 Fido barks to be let back inside.
   6.3 Todd or Gina hears Fido barking (again).
   6.4 Todd or Gina presses the button on the remote control.
   6.5 The dog door opens (again).
7. Fido goes back inside.

First Attempt: Wrong Approach

- In the new use case, we want to allow for the possibility that the bark recognizer hears Fido and opens the door before a human does
- It would be natural to take this approach at first
- 2. Todd or Gina hears Fido barking
    - 2.1 The bark recognizer "hears" a bark
- 3. Todd or Gina presses the button on the remote control
    - 3.1 The bark recognizer sends a request to open the door
- What's the problem with this approach?

Alternate Paths

- Recall that alternate paths are meant to show steps that can be done if something goes wrong with the current step
- In the original use case, steps 6.1 to 6.5 show another way in which the use case can move forward if the door closes before Fido is "done"
- They, in essence, document an **ADDITIONAL** set of steps that can occur between step 6 and step 7 of the "main path"
- The alternate path on the previous slide is different: 2.1 and 3.1 are meant to **REPLACE** steps 2 and 3 of the main path
    - Likewise for steps 6.3.1 and 6.4.1 (not shown) that are meant to replace steps 6.3 and 6.4
- Fortunately, there are no "hard and fast rules" in analysis. So, lets change the format of our use case a bit.

Use Case Evolved

What the Door Does

Main Path

1. Fido barks to be let out.
2. Todd or Gina hears Fido barking.
3. Todd or Gina presses the button on the remote control.
4. The dog door opens.
5. Fido goes outside.
6. Fido does his business.
    6.1 The door shuts automatically
    6.2 Fido barks to be let back inside.
    6.3 Todd or Gina hears Fido barking (again).
    6.4 Todd or Gina presses the button on the remote control.
    6.5 The dog door opens (again).
7. Fido goes back inside.

Alternate Paths

2.1 The bark recognizer “hears” a bark.
3.1 The bark recognizer sends a request to open the door.

6.3.1 The bark recognizer “hears” a bark (again).
6.4.1 The bark recognizer sends a request to the door to open.

Cool!

- This new way of showing the use case makes the purpose of alternate paths clear:
    - if the alternate path represents additional steps, we can keep them “in-line” with the main path
    - if the path represents replacement steps, we can show them off to the side
- One more problem
    - Our “main path” has our humans doing all the work
    - But the point of the change request was that they didn’t like that responsibility
        - If our bark recognizer succeeds, its going to be doing most of the work

Use Case Evolved (again)

What the Door Does

Main Path

1. Fido barks to be let out.
2. **The bark recognizer "hears" a bark.**
3. **The bark recognizer sends a request to open the door.**
4. The dog door opens.
5. Fido goes outside.
6. Fido does his business.
   - 6.1 The door shuts automatically
   - 6.2 Fido barks to be let back inside.
   - **6.3 The bark recognizer "hears" a bark (again).**
   - **6.4 The bark recognizer sends a request to the door to open.**
   - 6.5 The dog door opens (again).
7. Fido goes back inside.

Alternate Paths

2.1 Todd or Gina hears Fido barking.

3.1 Todd or Gina presses the button on the remote control.

6.3.1 Todd or Gina hears Fido barking (again).

6.4.1 Todd or Gina presses the button on the remote control.

What's a Scenario?

- Important Concept
  - A **complete path** through a use case, from the first step to the last, is called a **scenario**
  - Most use cases have **multiple scenarios** but a **single user goal**
    - Each use case has a single goal its trying to achieve, all paths through the use case attempt to achieve victory: meeting the goal
- In our use case, there are two variables
  - Does Fido get stuck outside?
  - Who hears Fido barking and opens the door?
- This leads to **seven** possible paths through our use case!

The Seven Paths (well, almost)

What the Door Does

Main Path

1. Fido barks to be let out.
2. The bark recognizer “hears” a bark.
3. The bark recognizer sends a request to open the door.
4. The dog door opens.
5. Fido goes outside.
6. Fido does his business.
    6.1 The door shuts automatically
    6.2 Fido barks to be let back inside.
    6.3 The bark recognizer “hears” a bark (again).
    6.4 The bark recognizer sends a request to the door to open.
    6.5 The dog door opens (again).
7. Fido goes back inside.

Alternate Paths

2.1 Todd or Gina hears Fido barking.
3.1 Todd or Gina presses the button on the remote control.

6.3.1 Todd or Gina hears Fido barking (again).
6.4.1 Todd or Gina presses the button on the remote control.

Ready to Code?

- Not quite!
  - We need to update our requirements list… how?

Ye Old Requirements List

1. The dog door opening must be at least 12” tall.

2. A button on the remote control toggles the state of the door: it opens the door if closed, and closes the door if open.

3. Once the dog door has opened, it should close automatically after a short delay (take that Rabbit!)

New Requirements

New Requirements List

1. The dog door opening must be at least 12” tall.

2. A button on the remote control toggles the state of the door: it opens the door if closed, and closes the door if open.

3. Once the dog door has opened, it should close automatically after a short delay (take that Rabbit!)

4. A bark recognizer must be able to tell when a dog is barking.

5. The bark recognizer must open the dog door when it hears barking.

Now we code!

- No problem
  - We create a new BarkRecognizer task
    - We have it point at an instance of the DogDoor
      - Just like the Remote class currently does
        - Indeed, they both point at the SAME instance of DogDoor
  - We update our code such that it invokes the recognizer when Fido barks
    - Our test code no longer shows Todd/Gina doing anything
- We compile/run and what happens?
  - Demonstration

Problem: The door doesn't close!

- Why?
  - Because the responsibility for closing the door in the original system was assigned to the Remote class
    - Seemed like a good idea at the time!
- So, how about we just copy the Timer code from Remote to BarkRecognizer
  - No problem, right?
- But, now we've got the responsibility for closing the door in Remote AND BarkRecognizer
  - AND, we've got duplicated code to boot… yuck!
- Where should the responsibility lie?
  - The DogDoor! It should take care of closing itself… and this eliminates the need for duplicating auto-door-closing code across multiple classes

Design Heuristic (to be made a principle later)

- **Duplicated code is bad**
  - How to remove?
  - The duplicated code is most likely duplicating behavior
    - If two classes behave the same, find some way to merge the behavior into a single class
- In the example, both the Remote and BarkRecognizer needed to make sure the door closed after they had opened it
  - We removed the need to do this by moving the behavior to the class they both shared, DogDoor
  - This makes semantic sense as well: DogDoor **SHOULD** be in charge of opening and closing the door, regardless of the context

Wrapping Up The Chapter

- **Change is constant** and your system **should always improve** every time you work on it
  - Sometimes a change in requirements reveals problems with your system that you didn't know were there
  - In the example, a new requirement revealed that a **responsibility** of the system **was assigned to the wrong class**
- More tools for the tool box
  - Requirements Principle: Your requirements **will always** change and grow over time
- OO Heuristic: **Duplicated code is bad**
  - remove the need for duplication by merging shared behaviors

But wait… Use Case Style Guidelines

- We've been talking about use cases without really discussing how to write them
- Fortunately, we have the work of Alistair Cockburn to draw on
- The next few slides are drawn from
  - Writing Effective Use Cases
  - by Alistair Cockburn
  - ISBN: 0-201-70225-8
- They present a "style guide" for writing the steps that appear in a use case
  - Cockburn calls the steps of a use case, **action steps**

Writing Action Steps

- Action Steps are written in one grammatical form
  - a simple action in which one actor either
    - accomplishes a task
    - or passes information to another actor
- Examples
  - User enters name and address
  - At any time, user can request the money back
  - The system verifies that the name and account are current

Action Step Guidelines

- #1: Use Simple Grammar
  - Subject…verb…direct object…prepositional phrase
    - The subject is important, see guideline 2
  - The system…deducts…the amount…from the account
- Bad writing makes the story hard to follow
- Complex writing makes it hard to extend an action step
  - e.g. if a step does three things, then if you extend that step, which “thing” does it extend?

Action Step Guidelines

- #2: Show Clearly “Who Has the Ball”
  - For each step, who is performing it?
    - Think of friends kicking a soccer ball
      - You can pass it to yourself
      - You can pass it to a friend
      - You can do something with the ball (e.g. perform a trick)
    - The person with the ball represents the actor
    - The ball represents information being passed between actors
      - You can manipulate the information or pass it on
  - At the end of the step, **who has the ball?**
    - The answer should always be clear in the writing

Action Step Guidelines

- #3: Write From a Bird's Eye View
  - Developers tend to write action steps from the system's perspective rather than a user's external perspective
    - e.g. "Get ATM Card and PIN" -- bad
    - rather "The customer inserts the card"
    - and "The customer enters the PIN"
  - Alternative Style
    - Customer: Inserts the Card
    - Customer: Enters the PIN

Action Step Guidelines

- #4: Show the Process Moving Forward
  - The amount of progress made in one action step varies according to the level of the use case
    - In high-level use cases, each step might satisfy a customer goal
    - In a low-level use case, each step may correspond to a computation by the system or data entry by the user
  - If a use case has more than 15 steps, it may indicate that the scope of each step is too small
    - Not “User hits tab key” but “User enters Name”
  - To find a slightly larger scope for a step, ask “Why is the actor doing this?” The answer is probably the scope you are looking for

Action Step Guidelines

- #5: Show the Actor’s Intent, Not the Movements
  - Before
    - System asks for name; User enters name
    - System prompts for address; User enters address
    - User clicks “OK”
    - System presents user’s profile
  - After
    - User enters name and address
    - System presents user’s profile

Action Step Guidelines

- #6: Include a “Reasonable” Set of Actions
  - Ivar Jacobson’s notion of a transaction
    - Actor sends request and data to system
    - System validates the request and data
    - System alters its internal state
    - System responds to actor with result
  - An action step can contain all four; or start with some in one step and end with the others in the subsequent step

Action Step Guidelines

- #7: “Validate” Do not “Check Whether”
  - Before
    - The system checks whether the password is correct
    - If it is, the system presents the available actions for the user
  - After
    - The system validates the password is correct
    - The system presents the available actions for the user
  - With “Checks” you always have to say “If true” or “If false” in the next step…not good; with validates you decide what actions go in the main path (or true branch) and then write the false branch as an alternate path

Action Step Guidelines

- #8: Optionally Mention the Timing
  - Most steps follow directly from the previous one
  - Occasionally you will need to say something like:
    - At any time between steps 3 and 5, the user will…
    - As soon as the user has …, the system will …
  - Feel free to put in the timing, but only when you need to
    - usually the timing is obvious

Action Step Guidelines

- #9: Idiom: "User has system A kick System B"
  - Situation: you need your system (A) to fetch information from another system (B)
  - Remember to keep the user in control
    - Not: User clicks Fetch button, at which time the system fetches data from system B (see #5)
    - But: User has the system fetch data from system B
  - Ball is clearly passed from user to A to B
  - responsibilities are clear
  - interface is not specified

Action Step Guidelines

- #10: Idom: “Do Steps x-y until Condition”
  - Situation: need to repeat a set of steps
  - If only one step needs repeating, put the repetition in the step
    - The user selects one or more products
  - If more than one step needs repeating, you can place the repetition before or after the set of steps; Cockburn recommends after in general, but before if the steps can occur in random order
    - See examples next slide

Action Step Guidelines

- Example: Putting Repetition Before
  - Customer logs into system
  - System presents products and services
    Steps 3-5 can happen in any order
  - User selects products to buy
  - User specifies form of payment
  - User specifies destination address
  - User finishes shopping
  - System processes order (of selected products with form of payment and ships to destination address)

Action Step Guidelines

- Example: Putting Repetition After
  - Customer supplies id or email address
  - System displays customer's preferences
  - User selects an item to buy
  - System adds item to customer's "cart"
    - Customer repeats steps 3 and 4 until done
  - Customer purchases the items in the cart

Wrapping Up

- The requirements of a system will always change
  - No matter how good the design of the system is
- We can deal with this constant pressure to change by working hard to have
  - a clear set of requirements
  - a good set of use cases
- If a change request comes in, we can
  - modify an existing use case or create a new one that shows how the system would behave after the change request is done
  - update requirements based on the new information
- Since use cases are so important, we reviewed ways to write good use cases

Ken’s Corner

- New Question and Answer site for programmers
  - <http://stackoverflow.com/>
    - Attempts to avoid common problems with looking up programming questions on-line: lots of “me too” posts, answers buried in long discussion, answer out-of-date, etc.
    - Web 2.0 features: answers are voted on allowing authors to gain reputation; participants are encouraged to edit original posts rather than engaging in long discussion; answers sorted by votes, etc.
- New site for hosting open source projects: Project Kenai
  - <http://kenai.com/>
- Site to support team-based projects: Assembla
  - <assembla.com>: consider using for framework/semester projects

Coming Up Next

- Lecture 8: Ready for the Real World
    - Read Chapter 4 of the OO A&D book
- Lecture 9: Nothing Stays the Same
    - Read Chapter 5 (part 1) of the OO A&D book