Discovering and Documenting Classes

Steven Zeil

September 10, 2013

Contents

1 Classification

Classification

Classification: Where do Classes Come From?
A key step in any OO analysis or design is identifying the appropriate classes.

- In practice, this process is
  - incremental
    We tend to add a few classes at a time.
  - iterative
    We may revisit earlier decisions and change them.
    We often identify classes and then later add details about their relationships to other classes.

. . . . . .

Grouping Objects into Classes

We may identify groups of objects as a class because they have common

- properties,

  e.g., we regard all things that have titles, authors, and textual content as *documents*, regardless of whether they are in a print medium, a file, or even chiseled into a set of stone tablets.

  - Don't make the mistake of grouping things into a class because they have common property *values*.
    - A "collection of documents" can be a class. The values of that class can be collections that were selected by many different criteria.
    - By contrast, the collection of documents written by Mark Twain (i.e., whose *author* property has the value *MarkTwain*) is not a class. It's just a particular value of the "collection of documents" class.

- behaviors,

  e.g., the set of all documents that can be loaded from and saved to a disk might represent a distinct class *ElectronicDocuments*.

. . . . . .

2 Driving the classification process

Driving the classification process

Where do we get the information from which we can identify classes during analysis?

The "program as simulation" philosophy suggests that we should be looking for a model of the "real world".

- Initially we will build that model by looking at informal English descriptions of the world.
- Later, from use cases (scenarios)

. . . . . .

Our first pass is often informal, based on the documents at hand.

We will eventually formalize this process by writing, in conjunction with our domain experts, use-cases (scenarios) of sequences of actions in the world and analyzing those scenarios to see what they suggest about classes of objects in the world and the responsibilities of those classes.

Working from Informal Descriptions

Generally, this is done at the start of construction of a *domain model* or, if no domain model is needed, of the *analysis model*.

A fairly simple way to get started is to scan the informal statement looking for noun phrases and verb phrases

- Nouns represent candidate objects
- Verbs the messages/operations upon them (responsibilities)

This doesn't scale well to large projects/documents, but it is simple and often a useful starting point.

After that, we move on by exploiting our knowledge to assign the responsibilities to the appropriate and to classes, refine our choice of classes and responsibilities.

. . . . . .

Use-Case analysis

A *use-case* is a particular pattern of usage, a scenario that begins with some user of the system initiating a transaction or sequence of related events.

We analyze use-cases to discover the

- objects that participate in the scenario
- responsibilities of each object
- collaborations with other objects

More on this in later lessons.

. . . . . .

3 Informal Documentation: CRC Cards

CRC Cards

Early in our development process, we won't want to be slowed down by the need to construct detailed documentation that looks "pretty" enough to show people outside our team (management or domain experts).

CRC cards are a popular way of capturing early classification steps.

. . . . . .

CRC

CRC (Class, Responsibility, & Collaborators) cards are a useful tool during early analysis, especially during team discussions.

They are not exactly a high-tech tool:

- 4x6 index cards
- used to take notes during analysis,
- as a concrete symbol for an object during disucssion
  - cards can be stacked, moved, etc. to illustrate proposed relationships

. . . . . .

A low-tech (no-tech?) approach is often useful in early brainstorming sessions. The index cards can serve as a concrete symbol for object during discussion. People trying to make a point may stack the cards, move them around, etc., while discussing proposed relationships.

CRC Cards are Informal Documentation

- The point of CRC cards is to capture info about an analysis discussion without slowing down that discussion so someone can take nicely formatted notes.
- They aren't pretty.
  - They aren't something you ever want to show your customers or even your own upper-management.
  - If you come out of a group meeting and your CRC cards aren't smudged, dog-eared, with lots of scratched-out bits, you probably weren't really trying.

. . . . . .

The point of CRC cards is to capture info about an analysis discussion without slowing down that discussion so someone can take nicely formatted notes. (Every now and then I see someone announce a new online tool for letting you type into a PC to write CRC cards that will then be printed out in nice

neatly formatted output. That really misses the point. Ever been in a meeting where some single person was trying to type up all the important stuff being said? What does that usually do to the discussion dynamic?)

3.1 CRC Card Layout

CRC Card Layout

- labeled with class name
- divided into two columns
  - *responsibilities*
    A high-level description of a purpose of the class
    * attributes
    * behaviors
  - *collaborators* other classes with which this class must work with (send messages to) to fulfill this class's responsibilities

ClassName	
responsibility 1 responsibility 2 responsibility 3	collaborator 1 collaborator 2

CRC Card Example

Librarian	
Handles checkout and checkin of publications	Patrons
Reshelves books	Publications
Manages new acquisitions of publications	Inventory
Has name, ID#, branch assignment	Catalog

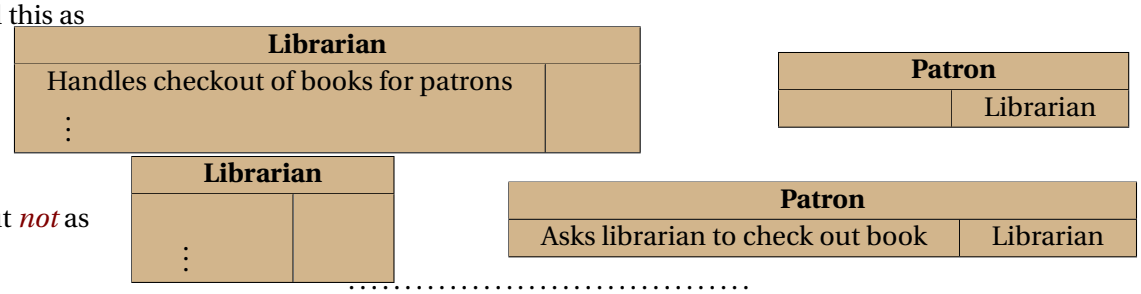
3.2 Assigning Responsibilities

Assigning Responsibilities

- The responsibilities will eventually evolve into messages that can be sent to this class and then into member functions of an ADT.
- Being aware of that intention can be a good indicator of which class should receive a particular responsibility.

Assigning Responsibilities Example

For example, if I were told "a library patron will give a librarian a book to be checked out", I would model this as



but *not* as

When A does B to C

A useful rule of thumb is that if "A does B to C", then

- "doing B" is a responsibility.
- But it is *usually* a responsibility of class C, not of A.
- C is then a collaborator of A

Natural language being the flexible and imprecise tool that it is, there are many exceptions to this. But that's where thinking of responsibilities as future functions can help. In programming terms, statements like "A does B to C" often occur in context where we are describing a series of steps being enacted because someone or something else asked A to fulfill some higher-level responsibility. In pseudo-code terms, we might say

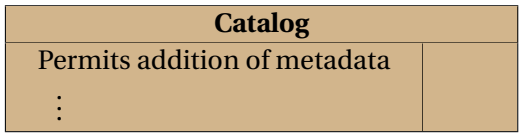
```
void A::fulfillSomeOtherResponsibilityOfA(C c1)
{
   ⋮
  c1.B();
   ⋮
}
```

in which case it is clear that B is a responsibility of class C, and C should be listed as a collaborator on A's card (along with that "some other responsibility").

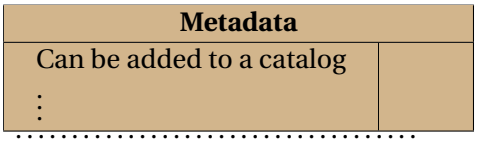
Containers

A variation on this rule of thumb occurs when managing collections.

If I told you that "the librarian adds the book's metadata[1] to the catalog", I would expect you to model that as



and not as



Containers (cont.)

Again, by analogy with programming, we understand that if you had something like

```
vector<int> v;
set<int> myList;
   ⋮
v.push_back(23);
myList.insert (23);
```

You would regard the ability to push data onto the end of a vector or to add data to a set as operations of the vector and the set, not as operations of the int data type.

You would never say:

```
23.insertInto (myList);
```

Not only is this ugly, but it suggests that we would need to predict all the possible container classes that will ever be written to hold integers, and to somehow add their specific add/insert/push operations to the set of permitted operations of the *int* type.

Similarly, we'd expect that metadata could be added to many other kinds of containers as well as the library catalog. The ability to hold multiple instances of metadata is a responsibility of the container, not of the thing contained.

3.3 Attributes

Attributes

A common variation (used in the textbook) is to use the backs of the cards to list attributes.

We won't do that. We'll list known attributes among the responsibilities,

[1] *Metadata* refers to the set of properties that identify and describe a document or other collection of data. Typical metadata fields are author, title, date of publication, etc. "Metadata" is a perfect example of the type of specialized vocabulary that the people working in the Library World would understand and that you as a software developer assigned to that world would need to learn to use when communicating with them.

- By convention, "Has" introduces an attribute or list of attributes.

Librarian	
Handles checkout and checkin of publications	Patrons
Reshelves books	Publications
Manages new acquisitions of publications	Inventory
Has name, ID#, branch assignment	Catalog

Attribute Conventions

- Singular and plurals are used in noun phrases to distinguish between single occurrences ("name") and collections ("publications").
  - Avoid suggestions of specific data structures (e.g., "array of publications").
  - More generic terms “collection”, “sequence” (if ordered) are OK.

3.4 Empty Columns

Empty Columns

It’s OK for one or the other column to wind up being empty.

- An empty list of responsibilities is often associated with classes that are *actors* - they send messages to other classes but nothing in this world sends messages to them.

  These often correspond to external systems or people who are acting spontaneously to initiate some action.

- An empty list of collaborators can indicate that a class is a *server* - it accepts messages but sends out none. This is most likely in a "lower-level" class that is little more than a collection of attributes. Most systems have many such classes.

- If both columns remain empty, however, that’s a good sign that the class may not be needed in your model.
  - Be careful, though, about jumping to this conclusion too soon. Dropping classes form the mode should only be done when you are nearing the completion of the model.

3.5 Common Mistakes in CRC Cards

Common Mistakes in CRC Cards

- Premature design: You aren't doing code, selecting data structures, etc., yet. If you *were* doing those, you shouldn't be using CRC cards.
- Overly specific collaborators: There is no significance between the vertical match up of the responsibilities and the collaborators, There's no implication that a particular collaborator goes with a responsibility "on the same line".

  Consequently, there's no reason to list a collaborator twice.
- Mis-assigned responsibilities: already discussed
- Inconsistent collaboration: If you place a class B in the collaborator list of card A, then there needs to be some responsibility of B that it actually makes sense for A to call upon or make use of.

  In particular, if class B has an empty responsibilities column, it really can't appear as a collaborator of anything at all!

. . . . . .