CSci 658: Software Language Engineering Using CRC Cards (Class-Responsibility-Collaboration)

H. Conrad Cunningham

27 April 2018

Contents

Acknowledgement: These notes are based, in part, on material from chapters 3, 4, 5, and 6 of [Bellin 1997].

Advisory: The HTML version of this document requires use of a browser that supports the display of MathML. A good choice as of April 2018 is a recent version of Firefox from Mozilla.

Objectives

- Introduce concept of CRC card [Beck 1989] [Bellin 1997]
- Discuss use in object-oriented analysis and design -- in particular Responsibility-Driven Design (RDD) [Wirfs-Brock 1990, 2003]
- To extract class abstractions from domain and problem requirements
- Use familiar case study: Automated Teller Machine (ATM)

What are CRC Cards?

- Provide informal, low-tech, low-cost, agile method for object-oriented analysis and design
- Use a physical (4 by 6 inch) index card to represent a class (or object) -- can use tools, but keep use of method lightweight
- Involve careful choice of names -- introduce vocabulary of system's domain and design
- Relate to other domain analysis tasks for software framework and domain-specific language design

CRC Card Preview

Class Name

description of responsibilities

Collaborators

other classes

What Goes On CRC Cards?

- Record three types of info on card
    1. **Class** name -- underlined upper-left corner -- noun
        - distinction between "object" (or "instance") and "class" fuzzy in early stages
    2. **Responsibilities** -- list on left side under name -- verb phrases
        - behaviors or operations
        - knowledge held
    3. **Collaborators** -- list on right side paired with responsibility

    - other classes used in carrying out responsibility

CRC Card Layout

Class Name

description of responsibilities

Collaborators

other classes

Who Writes CRC Cards (1)

- Acknowledge no one individual has all needed information
- Write CRC cards as team, rather than individually
- Choose team large enough to be diverse, but small enough to reach consensus

Who Writes CRC Cards (2)

- Form analysis team with five or six members
    - one or two user domain experts
    - one or two systems analysts
    - one object-oriented software designer
    - one team facilitator and leader
- Include experienced individuals
- Select facilitator skilled at OO & group techniques, not bossy
- Write CRC cards to focus team activities

Brainstorming

Use brainstorming session to collect ideas quickly and creatively

- Collect diverse set of ideas
- Compare ideas
- Synthesize unanticipated solutions

Brainstorming Principles

1. All ideas are potentially good ideas

   Don't censure yourself or others -- All ideas are equal

2. Think fast and furiously; ponder later

   Fast-paced discussion encourages individual creativity

3. Give every voice a turn

   Include everyone in group -- no loudmouth domination

4. A little humor can be a powerful force

   Humor can break barriers, relieve tension, and build trust

Candidate Class Sources (1)

Before session, assign each team member an investigative task

- Read *all* requirements documents
  - Examine any formal or informal requirements documents (e.g. user stories)
  - Don't overlook indirect sources -- memos, meeting minutes, etc.
  - Circle nouns, pronouns, and noun phrases -- potential classes
- Look carefully at reports
  - Examine reports generated in old (manual or automated) system
  - Examine profiles for reports desired in new system
  - Circle nouns, pronouns, and noun phrases -- potential classes

Candidate Class Sources (2)

- Conduct interviews
  - Talk to experienced users of old system
  - Record interview or take precise notes
  - Identify nouns, pronouns, and noun phrases -- potential classes
- Examine documentation and files
  - Review documentation on old system
  - Review any available unofficial/personal notes from users or maintainers
  - Identify nouns, pronouns, and noun phrases -- potential classes

A good analyst is a good detective!

Identifying verbs and verb phrases may help find responsibilities later

Landshark Bank ATM (1)

Landshark Bank, a new bank opening next summer, plans to provide a full service automated teller machine (ATM) system.

The ATM system interacts with the customer through a display screen, numeric and special input keys, a bankcard reader, a deposit slot, and a receipt printer.

Customers may make deposits, withdrawals, and balance inquires using the ATM machine, but the update to accounts will be handled through an interface to the Accounts system.

Customers are assigned a Personal Identification Number (PIN) and clearance level by the security system. The PIN can be verified prior to any transaction.

In the future, we plan to support routine operations such as a change of address or phone number using the ATM.

Landshark Bank ATM (2)

Landshark Bank, a new bank opening next summer, plans to provide a full service automated teller machine (ATM) system.

The **ATM system** *interacts* with the **customer** through a **display screen**, **numeric** and **special input keys**, a **bankcard reader**, a **deposit slot**, and a **receipt printer**.

Customers may *make* **deposits**, **withdrawals**, and **balance inquires** using the **ATM machine**, but the **update** to **accounts** will be *handled* through an **interface** to the **Accounts system**.

Customers *are assigned* a **Personal Identification Number (PIN)** and **clearance level** by the **security system**. The **PIN** can be *verified* prior to any **transaction**.

In the **future**, **we** plan *to support* **routine operations** such as a **change of address** or **phone number** using the **ATM**.

Brainstorming Steps (1)

1. Review brainstorming principles
2. State session objectives

- Have a precise objective -- clear to all, narrow enough to accomplish in session
- Avoid digression from objective

Brainstorming Steps (2)

3. Use a round-robin technique
    - Go from individual to individual
    - Individuals may “pass” if they have nothing to contribute
    - Stop when no one has anything to contribute

Brainstorming Steps (3)

4. Discuss and select
    - Restate objective
    - Organize candidate classes into 3 categories by consensus -- “winners” -- “losers” -- “maybes”
    - Discuss “maybes” for fixed time to decide if “winner” or “loser”
    - Postpone “maybe” items if more information needed to decide

Candidate Classes for ATM

ATM	FinancialTransaction	BankCard
BankCustomer	PIN	Account
SavingsAccount	CheckingAccount	Transfer
Withdrawal	Deposit	BalanceInquiry
Receipt	ReceiptPrinter	Keypad
Screen	CashDispenser	ScreenMessage
Display	FundsAvailable	DepositEnvelopeFailure
Balance	TimeOutKey	TransactionLog
Key	AccountHolder	Printer
ScreenSaver	Prompt	NumericKey

Identify Core Classes (1)

Divide candidate classes into categories in following order:

1. **critical classes** (“winners”), for which we will write CRC cards

Items that directly relate to main entities of application

ATM example: Account, Deposit, Withdrawal, BalanceInquiry

Identify Core Classes (2)

2. **irrelevant candidates** (“losers”), which we will eliminate

   Items that are clearly outside the system scope

   ATM example: Printer, ScreenSave, and Prompt -- related to user interface done later, but not to core banking application

Identify Core Classes (3)

3. **undecided candidates** (“maybes”), which we will review further

   Items that we cannot categorize without clarifying system boundaries and definition

   ATM example: What is an “ATM”?

CRC Card for Account, Phase 1

Account

description of responsibilities

Collaborators

other classes

Clarify System Scope

- Determine *scope* -- what is and what is not part of system
- Decide system boundary definitively
- Perhaps draw diagram to record system boundary

ATM example: What is scope of the ATM system?

- Does it handle everything -- banking application, user interface, and interactions between them?
- Is ATM responsible for updating accounting records? or just recording and mediating transaction activity?

Decision: Limit scope to banking information capture, leave user interface and account update to other systems

Identify Commonalities & Variabilities (1)

Commonality (*frozen spot*): part of system unlikely to change from one system variant to another [Coplien 1998]

Variability (*hot spot*): part of system likely to change from one system variant to another [Coplien 1998]

- Encapsulate hot spot inside component (apply *information hiding*)
- Design interfaces of and relationships among components so that they seldom change -- reflect commonalities

Identify Commonalities & Variabilities (2)

ATM example hot spot: Withdrawal handling

- Initial support for dispensing cash, but future support for updating cash cards
- Affected classes? Account, Withdrawal, FundsAvailable, BankCard

Use Appropriate Design Patterns

Design pattern: design structure successfully used in similar context in past -- reusable design

- Use appropriate general patterns -- e.g. from [Gamma 1995]
- Use appropriate local patterns that distill specific experiences of development organization on similar projects
- Add or modify classes to satisfy design pattern

ATM example: Interactions of ATM with outside entities might be modeled using local “system interaction pattern”

Result would be new core class AuthorizeSystemInteraction

Leverage Existing Frameworks

Software framework: collection of abstract and concrete classes that captures architecture and system operation

Extend framework classes to customize behavior

“Upside down library” -- system control resides in framework code that calls “down” to user-supplied code

Example: Graphical user interface toolkits like the Java Swing

- Potentially select appropriate software framework
- Add or modify core classes to fit framework

Eliminate Unneeded Core Classes (1)

- Remove ghost classes -- classes that, upon further examination, do not fit within application

  Look for classes outside scope of system

  ATM example: BankCustomer, Printer, and Keypad are relevant but outside scope

  System only needs to know about BankCustomer indirectly through BankCard info -- e.g. PIN

Eliminate Unneeded Core Classes (2)

- Combine synonyms -- use one name for items essentially same

  Different groups within organization may use different names for same thing

  ATM example: BankCustomer and AccountHolder probably synonyms. Adopt one or create new name.

- Be careful when same word refers to different things!

  ATM example: Balance and FundsAvailable for withdrawal may differ because of policy affecting account. Keep separate.

Distinguish Attributes & Classes (1)

Some candidate classes may represent information held by others

May be an attribute rather than a class if:

- It does not do anything -- has no operations

  ATM example: Balance and FundsAvailable have few meaningful operations -- both closely associated with Account

- It cannot change state

Distinguish Attributes & Classes (2)

Consider PIN from ATM example:

- If immutable, then perhaps make attribute of Account
- If can change state (valid, invalid, suspended) then should be class

Core Classes for ATM

- FinancialTransaction
- Account
- BalanceInquiry
- Withdrawal
- Deposit
- BankCard
- AuthorizeSystemInteraction

Undecided Classes for ATM

- BankCustomer (ghost -- with AuthorizeSystemInteraction)
- PIN (attribute)
- SavingsAccount (attribute of Account)
- CheckingAccount (attribute of Account)
- ATM (ghost -- system name)
- FundsAvailable (attribute)
- Balance (attribute)
- Amount (attribute?)
- AccountHolder (synonym)

Irrelevant Items for ATM

Outside scope (most part of user interface system)

- Transfer (not handled in first version)
- Receipt, ReceiptPrinter
- Keypad, Screen, CashDispenser
- ScreenMessage, Display
- DepositEnvelopeFailure, TimeOutKey
- TransactionLog, Printer
- ScreenSaver, Prompt
- Numeric Key, Key

Naming (1)

- Names of classes should be singular nouns beginning with capitals
- Names of responsibilities (operations) should be short sequences of words containing one verb
- Names of booleans should indicate meaning of the *true* value
- Names should be easily recognized by domain experts
- Names should be consistent within project (whole organization?)

Naming (2)

- Names should be unambiguous -- Use abbreviations with care!
- Names should use capitalization and underscores, but avoid digits
- Names should be short
- Names should be pronounceable (read them out loud)

Characteristics of a Good Class

- Has a good name (as defined above)
- Has responsibility for behaviors
- Has responsibility for remembering knowledge
- Is needed by other classes (collaborates)
- Actively participates in system

If class has limited (e.g. one) responsibility, then reconsider whether to keep it (e.g. to promote reuse) or combine with others.

Assigning Responsibilities (1)

- After core classes identified, assign responsibilities to each class
  - responsibilities for exhibiting behaviors
  - responsibilities for holding knowledge
- Write responsibilities on CRC cards for each core class
- Often intermix task of finding collaborators with finding responsibilities
- Use combination of brainstorming and role-playing of scenarios (below) to discover responsibilities

Assigning Responsibilities (2)

- Focus on the *what*, not the *how*

  Grady Booch: "When considering the semantics of classes and objects, there is the tendency to explain how things work; the proper response is 'I don't care.' "

  False semantic distinctions among responsibilities may rob us of opportunities to use inheritance and polymorphism.

Responsibility Detection (1)

1. Brainstorm first. Refine later.

   Use brainstorming to identify set of candidate responsibilities for core classes

   Verbs that occur in requirements documents etc. may indicate behaviors (operations) that must exhibited

   Include rather than exclude. Don't worry about duplication

   Refine lists later

   Name each carefully

Responsibility Detection (2a)

2. Think simple. Factor out complexity.

   If most responsibilities cluster in one or two classes, then probably not good OO design -- does not exploit polymorphism and encapsulation

   ATM example:

   - Might give most responsibility to Account -- strong "manager" procedure giving commands to weak "workers"
   - Account probably too inflexible and others too insignificant to reuse
   - Give each class distinct role -- make each well-defined, complete, cohesive abstraction -- increase reusability

Responsibility Detection (2b)

ATM example:

- Give class Withdrawal responsibility "Withdraw Funds" -- making potentially useful to any class needing withdrawals

  - Give class Account responsibility "Accept Withdrawal" -- done by Withdrawal collaborating with it
  - Factor out complexity by identifying repeatedly occurring specialized behaviors -- create appropriate new classes

ATM example:

  - Perhaps factor out capturing and responding to user requests Introduce new class Form with responsibility "ask user for information"

Responsibility Detection (3)

3. Use abstraction to advantage.
    - Build hierarchies of classes
    - Abstract essence of related classes by identifying common responsibilities -- same "what", different "how"
    - Make these polymorphic operations -- defined in *abstract base class* -- given specific responsibility in *concrete subclass*

    ATM example:

    - Note commonalities among Withdrawal, Deposit, etc.
    - Create abstract base class Transaction with abstract responsibility "execute a financial transaction"
    - Implement responsibility differently for each subclass

Responsibility Detection (4)

4. Do not marry one solution. Play the field first.
    - Remember CRC cards are inexpensive and erasable!!
    - Experiment with different configurations of classes and assignments of responsibilities
    - Change CRC cards early to avoid changing code later

CRC Card for Account, Phase 2

Account

Know balance

Accept Deposit

Accept Withdrawal

other responsibilities

Collaborators

other collaborators

Assigning Collaborations (1)

- Identify relationships among classes
  - Each class specializes in some set of knowledge and behaviors
  - Classes must cooperate to accomplish nontrivial tasks
  - Thus *collaborations* between classes important
- Pair collaborations with responsibilities on CRC card
- Use *scenario*-based role-play to find/test collaborations
  - Scenario -- system behavior and sequence of events to realize it -- use case, user story
  - Simulating execution enables team to discover appropriate responsibilities and collaborations

Assigning Collaborations (2)

- Add collaborations when relationships found
  - Identify clients and servers
    - Server -- class that provides a resource
    - Client -- class that uses a resource

    Server is collaborator of client, not vice versa

    ATM example: In Withdrawal operation, Account is server for Withdrawal client
  - Identify hierarchies of classes

    ATM example: Transaction as superclass of Withdrawal, Deposit, etc.

Hierarchy Identification Tips (1)

- Explore *is-a* (“kind-of”) relationships (possible inheritance)

  ATM – Withdrawal is a (“kind of”) Transaction

  ATM – Withdrawal is not “part of” Transaction
- Name key abstractions
- Separate mixed classes where necessary
- Place super/subclass sets in hierarchies

Hierarchy Identification Tips (2)

- Look for reusable behaviors
  - reuse existing patterns and frameworks
  - record new patterns and frameworks for future reuse

CRC Role Play Steps (1)

1. Create a list of *scenarios* for use of system (i.e. *use cases*, *user stories*, actions to realize requirement)

   Use brainstorming

   ATM example: customer withdraws cash
2. Assign roles of classes to team members

   Each member has one or more classes

CRC Role Play Steps (2)

3. Rehearse the scenario

   Execute scenario with team members announcing what affected classes are doing -- putting cards into play when used
4. Correct CRC card and revise scenario
5. Repeat above two steps as necessary until rehearsal smooth
6. Perform final scenario

Develop Role-Play Scenarios (1)

1. Concentrate on "must do" scenarios

   Core behaviors touch central features of system

   ATM example: customer withdraws cash

2. Develop conditional "can do if" scenarios

   Routine tasks carried out under certain conditions

   Tasks to avoid exceptional situations

   Be careful about crossing out of system scope

Develop Role-Play Scenarios (2)

3. Record "might do" scenarios ("abuse cases") to test flexibility

   Exceptions -- unusual, complex, difficult to handle cases

   Help uncover poor collaborations and clarify system scope (a stress test)

   ATM example: Withdrawal with insufficient funds

4. If larger than 25 scenarios in "must do" and "can do if" categories, break into subsystems

Effective Role Play (1)

- Stick to the scenario(s)
- Limit session time (<= 2 hours)
- Use conscious, deliberate problem-solving activities

Effective Role Play (2)

- Use conscious, deliberate problem-solving activities
  1. Warm up
     - Prepare every session for effective group work by breaking down inhibitions
     - Use games, puzzles, brainteasers, mock brainstorming on "fun" topic.
     - Use "go-around" review of activities since last session (use if organizational culture discourages above)
  2. Enactment

    - Act out each scenario
3. Assessment
    - Build lists of problems encountered
    - Evaluate problems found

Warm-Up Tips

- Warm-up time never wasted
- Warm-up every session
- Inspire confidence of all team members, especially for new or changed teams
- Don't get too serious too fast

Scenario Enactment (1)

- Identify and summarize scenarios to be enacted (agenda)
    - Handle "must do" first, then "can do if"
- Assign roles to actors
    - Distribute CRC cards
    - Collaborators to different persons
    - Use domain experts for classes if possible
    - Rotate assignments from session to session
- Initiate scenario -- hold up first class
    - ATM example: Start with Customer or BankCard

Scenario Enactment (2)

- Proceed from responsibility to its collaborators
    - Hold up active card or toss "in play" on the table
- Watch action to detect problems
- When errors discovered
    - Correct if minor
    - Take notes if complex
    - stop enactment immediately if significant and complex -- go to assessment

Scenario Enactment (3)

- Avoid unnecessary changes, especially after considerable role play
- Do enactment-assessment on multiple scenarios per session

Scenario Assessment

- "Go-around" comments
- Identify problem points
- Create problem-solving priority list
- Change or confirm CRC cards
- Identify any scenarios that need to be repeated

Wider Use

- Use similar analysis techniques to build broader model of domain concepts
- Use similar Scope-Commonality-Variability (SCV) analysis [Coplien 1998] to design software families (frameworks, product lines)

  encapsulate variabilities to allow customization
- Use SCV analysis in design of domain-specific language (DSL) syntax and semantics

Summary

- Introduced CRC cards – lightweight, informal
- Discussed how to use them in OO analysis and design
- Examined how to use them in teams

References (1)

[Beck 1989] Kent Beck and Ward Cunningham. A Laboratory for Teaching Object-Oriented Thinking, In *Proceedings of the OOPSLA'89 Conference*, ACM, 1989.

[Bellin 1997] David Bellin and Susan Suchman Simone. *The CRC Card Book*, Addison Wesley, 1997.

[Budd 2000] Timothy Budd. *Understanding Object-Oriented Programming with Java*, Updated Edition, Addison Wesley, 2000.

References (2)

[**Coplien 1998**] J. Coplien, D. Hoffman, and D. Weiss. Commonality and Variability in Software Engineering, *IEEE Software*, 15(6):37–45, November 1998. [local]

[**Gamma 1995**] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[**Wilkinson 1995**] Nancy M. Wilkinson. *Using CRC Cards: An Informal Approach to Object-Oriented Development*, Cambridge University Press, 1995.

References (3)

[**Wirfs-Brock 1990**] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*, Prentice-Hall, 1990.

[**Wirfs-Brock 2003**] Rebecca Wirfs-Brock and Alan McKean. *Object Design: Roles, Responsibilities, and Collaborations*, Addison-Wesley, 2003.