Validation: CRC Cards

Massimo Felici

CRC Cards

- **CRC: Class-Responsibility-Collaborator**
- CRC cards provide the means to validate the class model with the use case model
- **Responsibilities** are a way to state the rationale of the system design
- CRC cards support responsibility-based modelling

 ©2004-2010

Slide 1: CRC Cards

CRC cards allow a useful early check that the anticipated uses of the system can be supported by the proposed classes. They support a brainstorming technique that works with scenario walkthroughs to stress-test a design.

Responsibility-based modelling is appropriate for designing software classes as well as for partitioning a system into subsystems. The underlying assumptions are:

- People can intuitively make meaningful value judgements about the allocation of responsibilities
- The central issues surrounding how a system is partitioned can be captured by asking what the responsibility of each part has toward the whole - Is it really the responsibility of this object to handle this request? Is it its responsibility to keep track of all that information?

Slide 1: CRC Cards

Required Readings

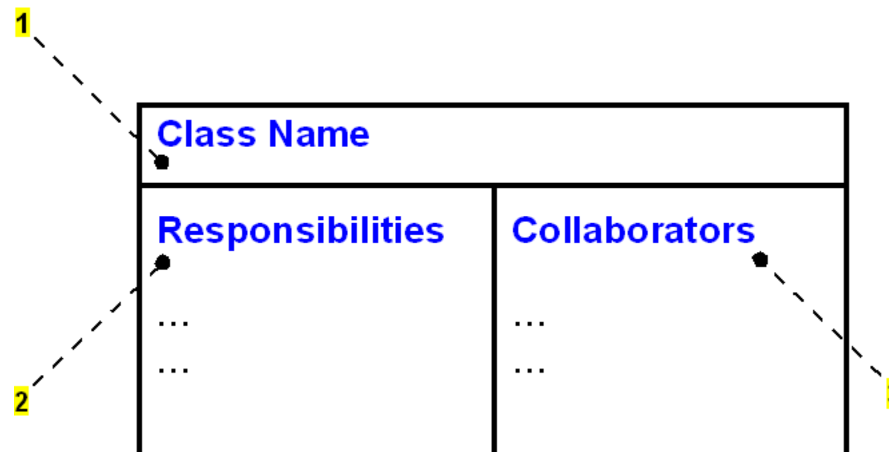
- K. Beck, W. Cunningham. A Laboratory for Teaching Object-Oriented Thinking. In Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '89), ACM, pp. 1-6.

School of informatics

CRC Cards

CRC Cards explicitly represent multiple objects simultaneously

1. The **Name** of the class it refers to
2. The **Responsibilities** of the class
   These should be high level, not at the level of individual methods
3. The **Collaborators** that help discharge a responsibility



 ©2004-2010

Slide 2: An empty CRC card

Class Name	
Responsibilities	**Collaborators**

Design by Responsibilities

- Responsibility-based Modelling allows
  - The identification of the components from which the system is constructed
  - The allocation of responsibilities to system components
  - The identification of the services (or functionalities) provided by them
  - The assessment how components satisfy the requirements as stated by the use cases
- Types of Responsibilities
  - To do something (active responsibilities)
  - To provide information (acting as a contact point)

 ©2004-2010

Steps in Responsibility-based Design

1. Identify scenarios of use; bound the scope of design
2. Role play the scenarios, evaluating responsibilities
3. Name the required responsibilities to carry a scenario toward
4. Make sure that each component has sufficient information and ability to carry out its responsibility
5. Consider variations of the scenario; check the stability of the responsibility
6. Evaluate the components
7. Ask the volatility / stability of the component
8. Create variations
9. Run through the variant scenarios to investigate the stability of the components and responsibilities
10. Simulate if possible
11. Consolidate the components by level
12. Identify subsystems
13. Identify the different levels
14. Document the design rationale and key scenarios
15. Decide which scenarios to document
16. List the components being used that already exist
17. Specify each new component

 ©2004-2010

Design Activities

1. **Preparation:** collection and selection of use cases
2. **Invention:** (incremental) identification of components and responsibilities
3. **Evaluation:** questions and scenarios stress test the design
4. **Consolidation:** further assessment of the tested components
5. **Documentation:** recording identified reasons and scenarios

CRC Cards in Design Development

1. Work using role play – different individuals are different objects
2. Pick a use case to building a scenario to hand simulate
3. Start with the person who has the card with the responsibility to initiate the use case
4. In discharging a responsibility a card owner may only talk to collaborators for that responsibility
5. Gaps must be repaid and re-tested against the use case

 ©2004-2010

Using CRC Cards

1. Choose a coherent set of use cases
2. Put a card on the table
3. Walk through the scenario, naming cards and responsibilities
4. Vary the situations (i.e., assumptions on the use case), to stress test the cards
5. Add cards, push cards to the side, to let the design evolve (that is, evaluate different design alternatives)
6. Write down the key responsibility decisions and interactions

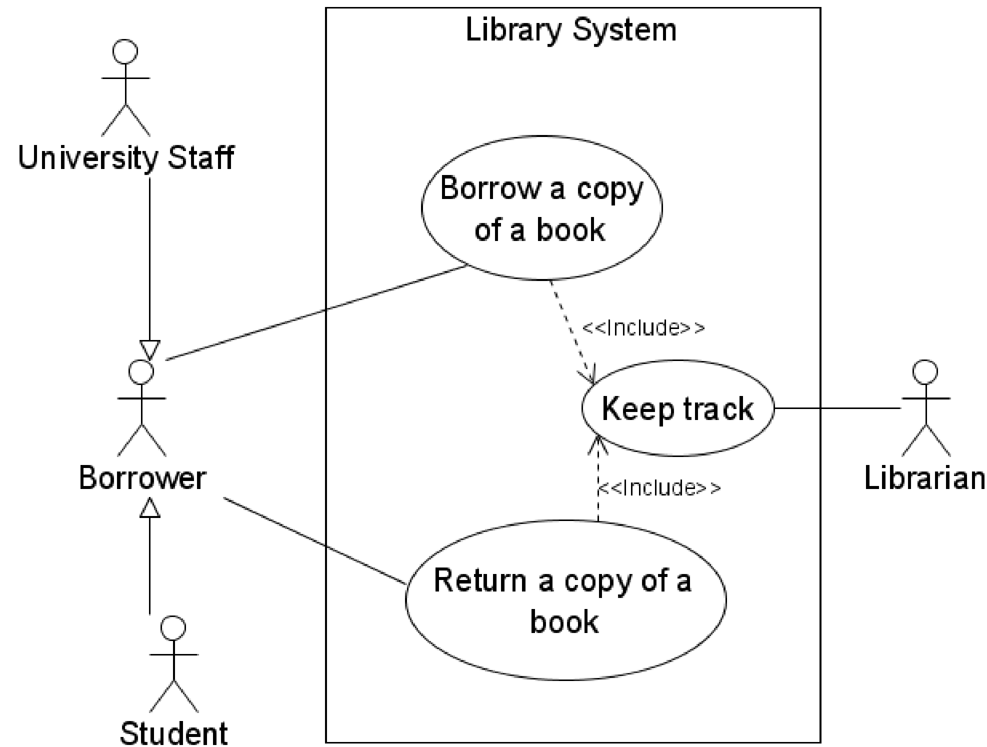
 ©2004-2010

A Library System

1. The library system must keep track of when books are borrowed and returned
2. The system must support librarian work
3. The library is open to university staff and students
4. University staff can borrow up to 25 different books
5. Students can borrow up to 15 different books

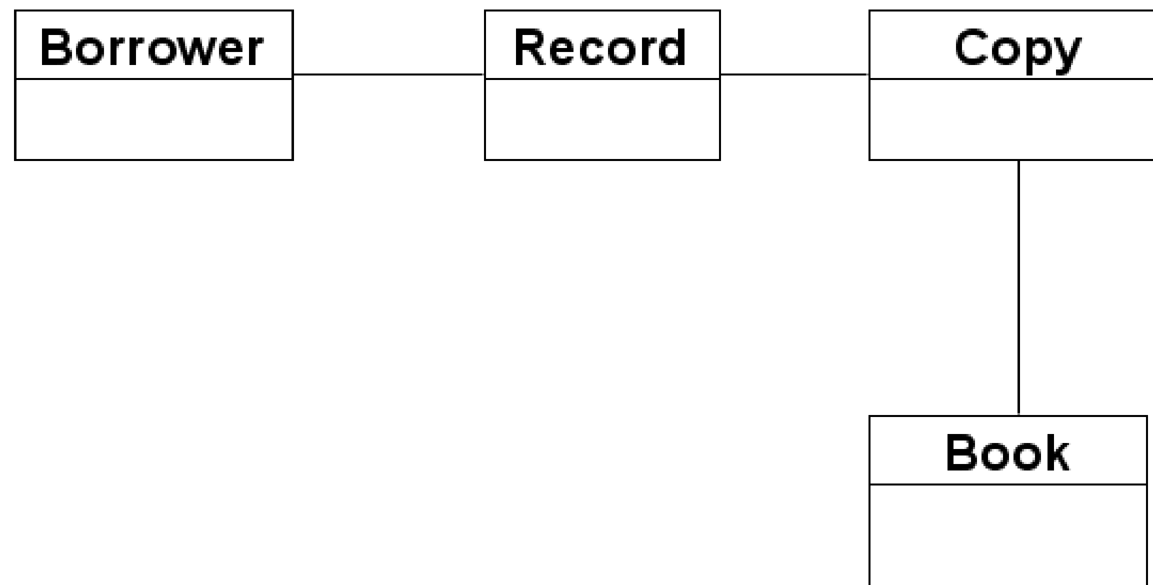
 ©2004-2010

A Library System



©2004-2010

A Library System



 ©2004-2010

A Library System

Borrower

Responsibilities	Collaborators
Borrow book	Record
Return Book	Record

Record

Responsibilities	Collaborators
Keep track	Copy

Copy

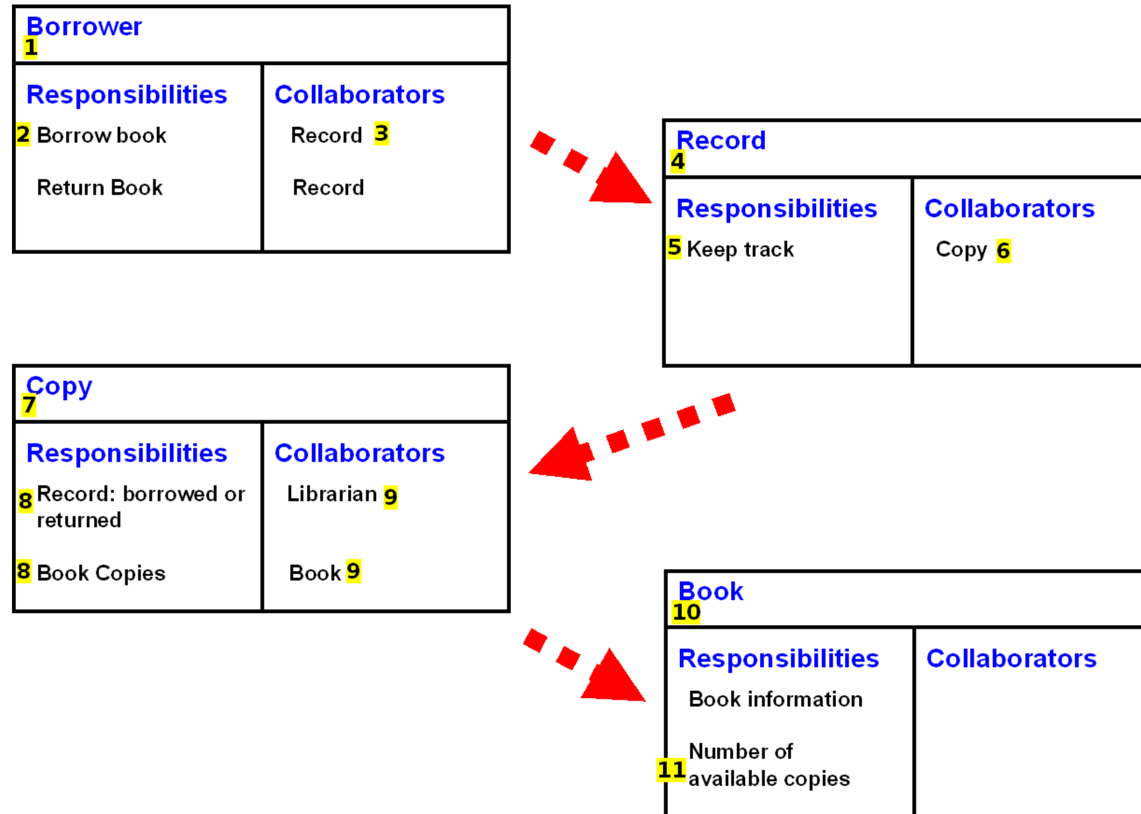
Responsibilities	Collaborators
Record: borrowed or returned	Librarian
Book Copies	Book

Book

Responsibilities	Collaborators
Book information	
Number of available copies	

 ©2004-2010

A Library System



Slide 12: A Library System

- Note that playing with CRC cards points out interactions between classes
- UML provides specific notations (e.g., communication or sequence diagrams) for modelling these interactions

What CRC Card help with

- Check use case can be achieved
- Check associations are correct
- Check generalizations are correct
- Detect omitted classes
- Detect opportunities to refactor the class model, that is, to move responsibilities about (and operations in the class model) without altering the overall responsibility of the system

CRC Cards and Quality

- CRC Cards
  - provide a good, early, measure of the quality of the system (design) – solving problems now is better that later
  - are flexible - use them to record changes during validation
- **Too many responsibilities**
  - This indicates low cohesion in the system
  - Each class should have at most three or four responsibilities
  - Classes with more responsibilities should be split if possible
- **Too many collaborators**
  - This indicates high coupling
  - It may be the division of the responsibilities amongst the classes is wrong

 ©2004-2010

Principles for Refactoring

- Do not do both refactoring and adding functionality at the same time
- Make sure you have good tests before you begin refactoring
- Take short deliberate steps

Slide 15: Principles for Refactoring

- Do not do both refactoring and adding functionality at the same time
  - Put a clear separation between the two when you are working
  - You might swap between them in short steps, e.g., half an hour refactoring, an hour adding new function, half an hour refactoring what you just added
- Make sure you have good tests before you begin refactoring
  - Run the tests as often as possible; that way you will know quickly if your changes have broken anything
- Take short deliberate steps
  - Moving a field from one class to another, fusing two similar methods into a super class
  - Refactoring often involves many localized changes that result in a large scale change
  - If you keep your steps small, and test after each step, you will avoid prolonged debugging

When to Refactor?

- When you are adding a function to your design (program) and you find the old design (code) getting in the way
- When you are looking at design (code) and having difficulty understanding it

 ©2004-2010

Slide 16: When to Refactor?

When adding a new function starts becoming a problem, stop adding the new function and instead refactor the old design (code). Refactoring is a good way of helping you understand the design (code) and preserving that understanding for the future.

Suggested Readings

- T. Mens, T. Tourwe. A survey of software refactoring. IEEE Transactions on Software Engineering, vol.30, no.2, pp. 126-139, February, 2004.

OO Design using CRC Cards

1. Review quality of class model
2. Identify opportunities for refactoring
3. Identify (new) classes that support system implementation
4. Identify further details (e.g., sub-responsibilities of class responsibilities, attributes, object creations, destructions and lifetimes, passed data, etc.)

 ©2004-2010

OO Analysis using CRC Cards

1. Session focuses on a part of requirements
2. Identify classes (e.g., noun-phrase analysis)
3. Construct CRC cards for these and assign to members
4. Add responsibilities to classes
5. Role-play scenarios to identify collaborators

 ©2004-2010

Common Domain Modelling Mistakes

- Overlay specific noun-phrase analysis
- Counter-intuitive or incomprehensible class and association names
- Assigning multiplicities to associations too soon
- Addressing implementation issues too early
  - Presuming a specific implementation strategy
  - Committing to implementation constructs
  - Tackling implementation issues (e.g., integrating legacy systems)
- Optimising for reuse before checking use cases achieved

 ©2004-2010

Readings

Required Readings

- K. Beck, W. Cunningham. A Laboratory for Teaching Object-Oriented Thinking. In Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '89), ACM, pp. 1-6.

Suggested Readings

- T. Mens, T. Tourwe. A survey of software refactoring. IEEE Transactions on Software Engineering, vol.30, no.2, pp. 126-139, February, 2004.

 ©2004-2010

Summary

- We should try to check the completeness of the class model (early assurance the model is correct)
- CRC Cards are a simple way of doing this
- CRC Cards support responsibility-based modeling and design
- CRC Cards identify errors and omissions
- They also give an early indication of quality
- Use the experience of simulating the system to refactor if this necessary

 ©2004-2010