Advanced Software Engineering

Chapter Seven

UML Modelling - Use Cases for Design

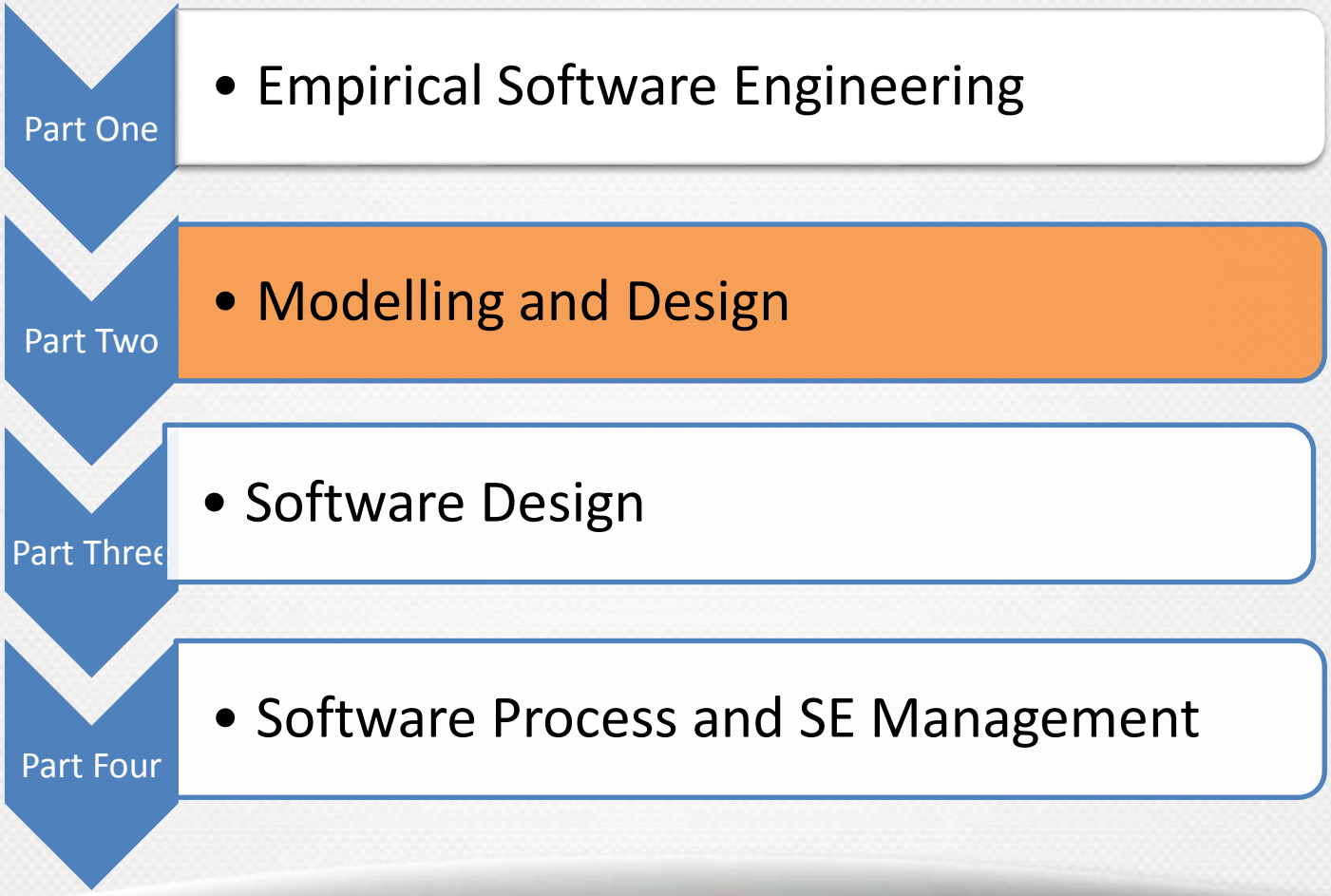
Learning Outcomes

Know role use cases for software design

Know how to use Essential Use Cases

Text:
Biddle Paper on Essential Use Cases
Larman Chapter 9 & 31

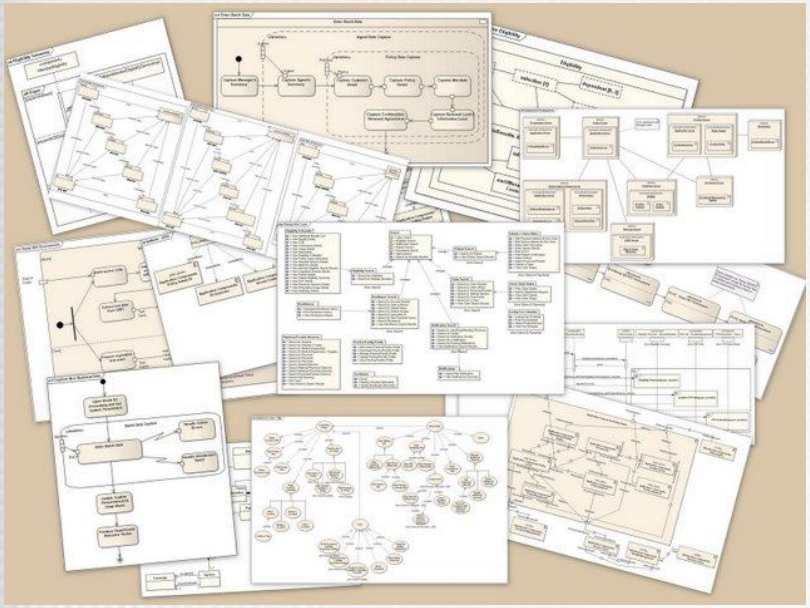
Module Structure

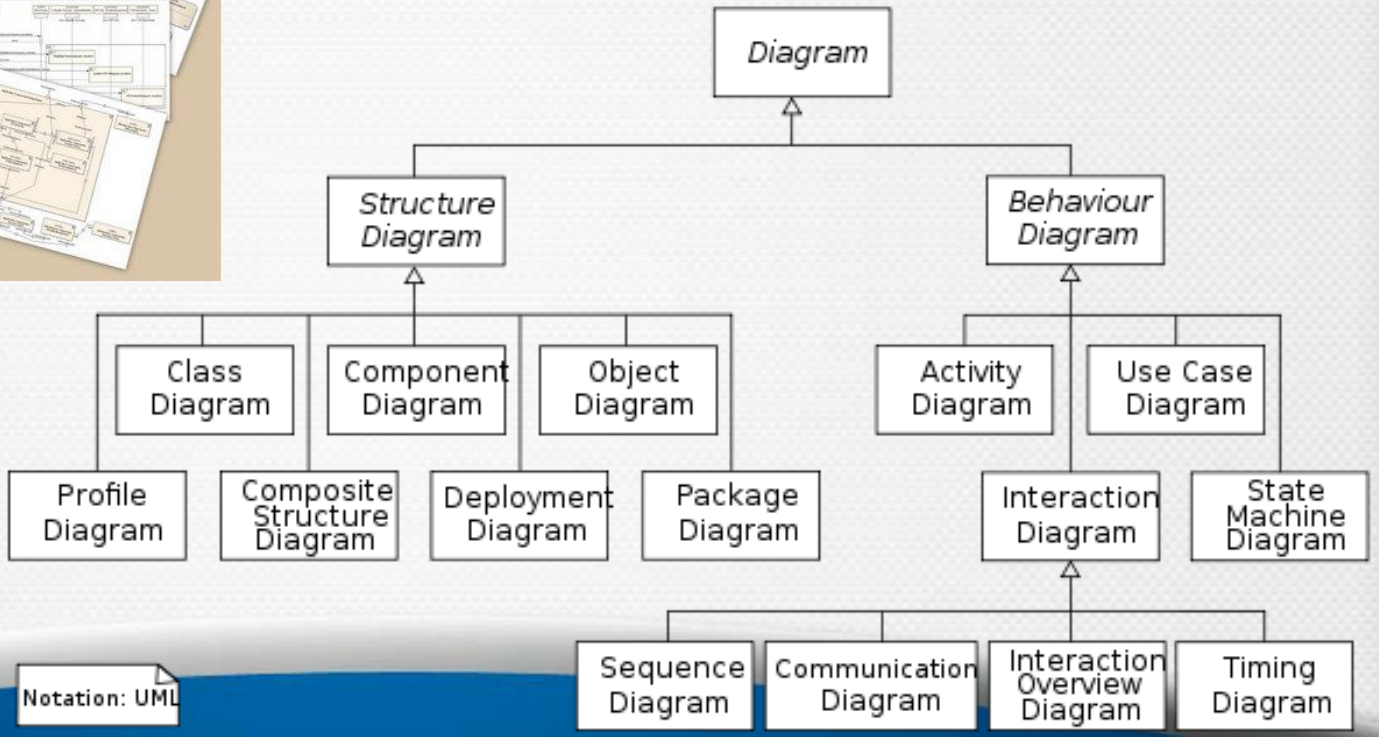


UML

- UML is a visual language for specifying, constructing and documenting the artefacts of systems (OMG – www.omg.org)
- How to use UML
  - Sketches – informal and incomplete useful for exploration and understanding
  - Blueprint – detailed diagrams for reverse engineering or forward engineering (code generation)
  - Programming language – “Executable UML” will be a reality soon

UML has 14 diagrams





Use Case Style

- Brief – terse one paragraph summary = main success scenario
- Casual – Informal paragraph format – multiple paragraphs covering various scenarios
- Fully dressed – all steps and variations in detail

- Alternatively – essential use cases (coming later)

Fully Dressed

Use Case Section	Comment
Use Case Name	Start with verb
Scope	The system under design
Level	“user goal” or “subfunction”
Primary Actor	Calls on the systems to deliver its services
Stakeholders and their interests	Who cares about this use case – what do they want?
Preconditions	Must be true on start
Success Guarantee (post conditions)	What must be true after successful completion
Main Success Scenario	Typical success path
Extensions	Alternative success/failure scenarios
Special Requirements	Related Non-functional requirements
Technology /Data Variations List	Varying I/O methods and data formats
Frequency of Occurrence	How often use case executes
Miscellaneous	Open issues

Essential Use Cases

- Abstract, lightweight, technology-free dialogues of user intention and system responsibility that effectively capture requirements for *user interface design*.
- Drive object-oriented development
  - Allow UI + OO development to proceed in parallel
- Assumes design will follow
  - Makes a start on design since it separates System Responsibilities from User Responsibilities

Going 'Essential'

gettingCash	
User Action	*System Response*
insert card	
	read magnetic stripe request PIN
enter PIN	
	verify PIN display transaction menu
press key	
	display account menu
press key	
	prompt for amount
enter amount	
	display amount
press key	
	return card
take card	
	dispense cash
take cash	

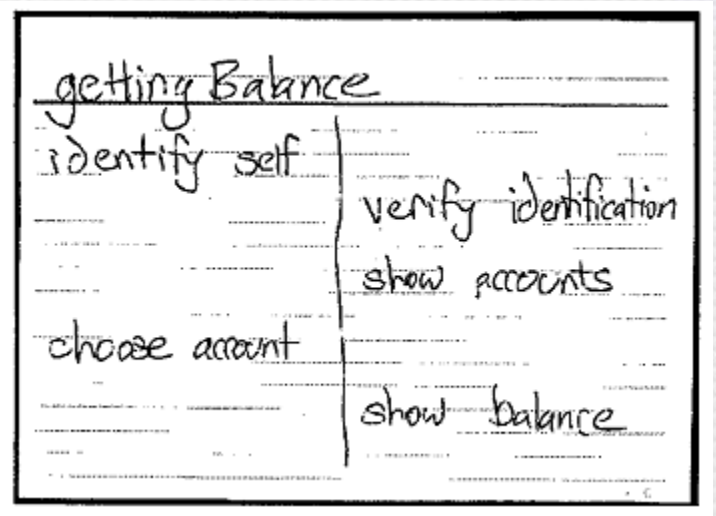
Traditional use Case – organised into User Action/ System Response columns

gettingCash	
User Intention	*System Responsibility*
identify self	
	verify identity offer choices
choose	
	dispense cash
take cash	

- Essential Use Case
  - User Action *abstracted* to User Intention
  - System Response *expanded* to Responsibility

System Responsibility

- System Responsibility means
  - expression of what needs to be done, without unnecessary detail of how it will be done
  - Abstraction
  - Prompts more detail of System internals than ordinary use case
  - First Step to design – see RDD later
  - Emphasize abstract behaviour
    - Nothing about implementation structure.
    - But NB: OO means objects fulfilling responsibilities via methods & messages
    - Responsibilities may also be factored to higher level abstract classes.



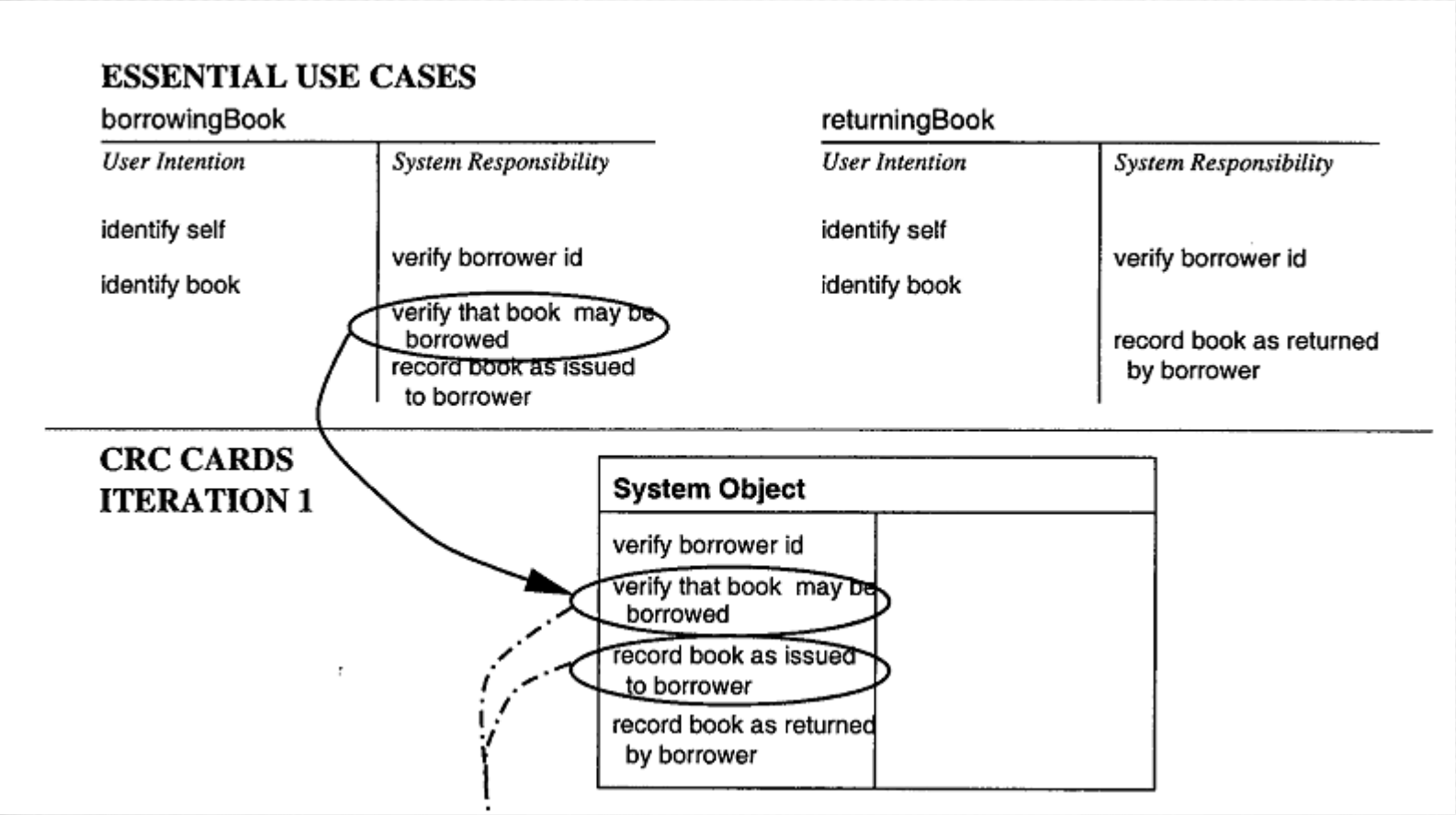
gettingCash

User Intention	*System Responsibility*
identify self	
	verify identity
	log transaction start
	offer choices
choose	
	dispense cash
	adjust balances
take cash	
	log transaction finish

Link to CRC cards

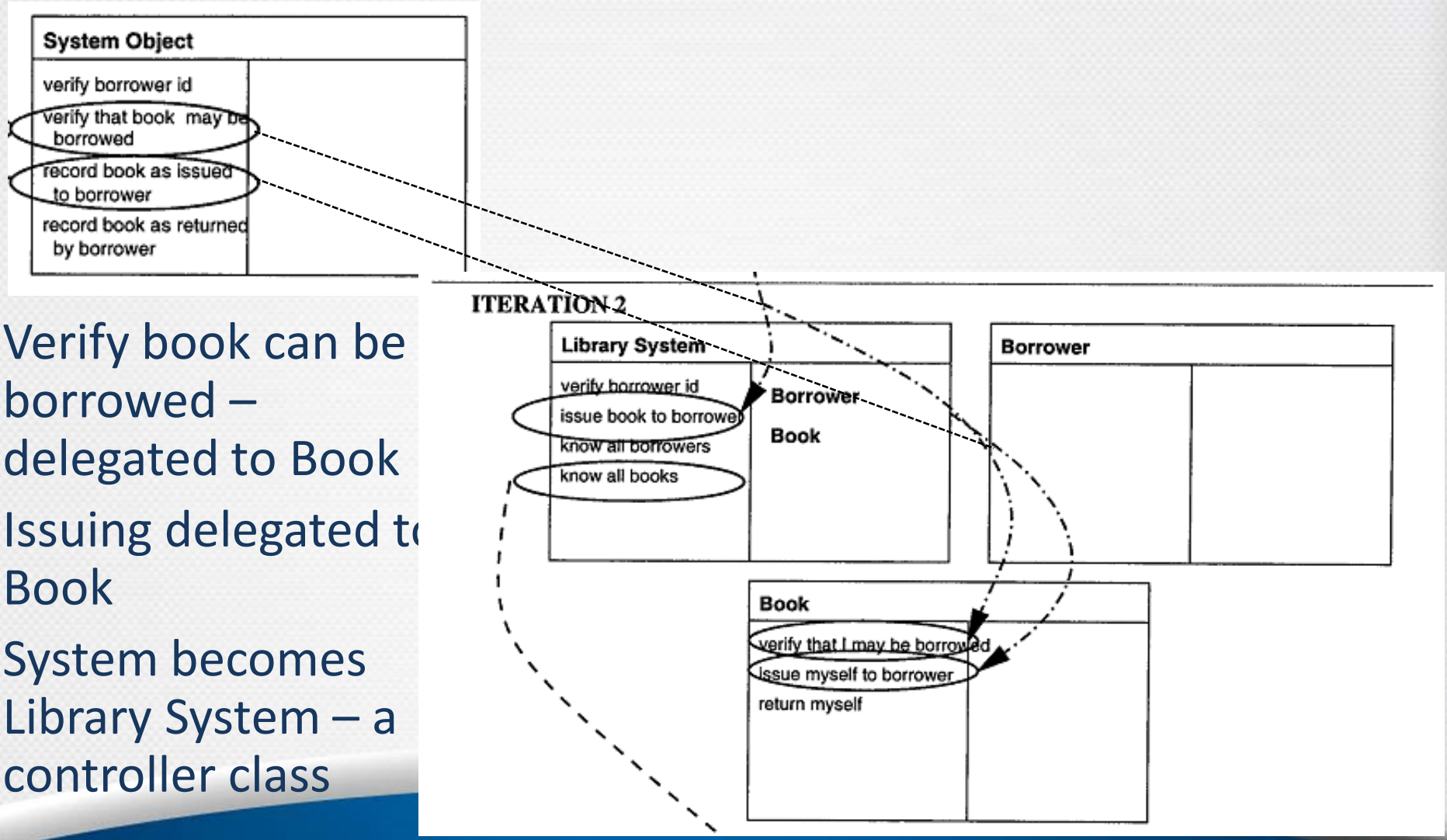
- Closely linked – Class-responsibility-collaboration cards can be derived from use cases
- May also have Super Class named
- May also have a list of subclasses

Example

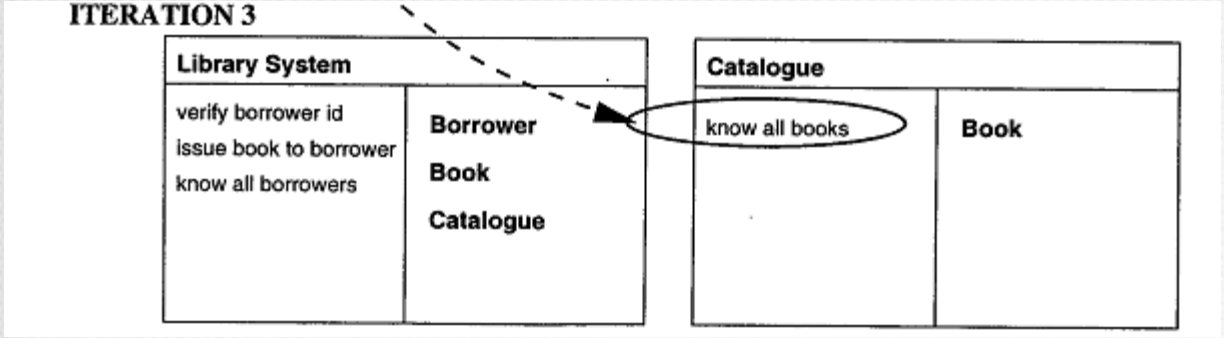


- Need some class(es) to encapsulate these responsibilities

Example – Iteration 2



- Verify book can be borrowed – delegated to Book
- Issuing delegated t Book
- System becomes Library System – a controller class



- Know all books was added to Library System in iteration 2
- Good design says factor this out as a Singleton – Catalogue
- Essential Reading on Eseential Use Cases – Biddle et al - 2008

Essential Use Cases and Responsibility in Object-Oriented Development

Robert Biddle, James Noble, Ewan Tempero
School of Mathematical and Computing Sciences
Victoria University of Wellington
Wellington, New Zealand
(robert,kjx,ewan)@mcs.vuw.ac.nz

Abstract

Essential use cases are abstract, lightweight, technology-free dialogues of user intention and system responsibility that effectively capture requirements for user interface design. We describe how essential use cases can also drive object-oriented development directly, without any intervening translation, and allowing user interface development and object-oriented development to proceed in parallel. Working with essential use cases yields some unexpected further benefits: the crucial common vocabulary of responsibilities lets designers trace directly from the essential use cases to the objects in their design.

Keywords: Use Cases, Object-Oriented Analysis, Object-Oriented Design, Traceability

1 Introduction

Use cases are now widely accepted for capturing requirements for object-oriented software development, and are commonly supported in modeling languages and in development processes, such as the Unified Modeling Language [19] and the Rational Unified Process [14]. Constantine and Lockwood's *Usage-Centered Design* [9] introduced *essential use cases* — use cases written specially to be abstract, lightweight, and technology-free — to support user interface design. Following a conventional process, essential use cases would be used to produce the user interface design, then, once that design was complete, essential use cases could be translated into more conventional use cases — much more concrete and detailed descriptions of a system and its interface design. Unfortunately, translating essential use cases to a more conventional form requires effort, costs time, and delays work on the object-oriented development until the user interface design is complete.

In this paper, we explore the application of essential use cases directly to object-oriented software development. We speculated that essential use cases would work just as well as conventional use cases as a starting point for object-oriented design; that technology independence could better support requirements gathering, because there would be less need to specify details that are only relevant to the design; and that the brevity of essential use cases would better support communication between developers and stakeholders.

We began using essential use cases as our prime requirements gathering tool, and have now used this approach for a number of system development projects over the last two years. We have found that essential use cases are suitable for object-oriented software development in general, and indeed have significant advantages over conventional use cases.

We have three general results to report. The first is what we speculated: that essential use cases can drive object-oriented design directly, without first writing more concrete conventional use cases. The other results were unanticipated. One is that essential use cases provide practical, operational guidance on how to move to an object-oriented design from the requirements. Another is that responsibilities provides a common vocabulary that supports seamless traceability forwards and backwards between essential use cases and objects. A related finding is that, because essential use cases show only the bare essentials of the use case, we are able to identify patterns of use cases, which can be used to make the requirements gathering more efficient. We have addressed this result elsewhere [5].

This paper is organized as follows. We begin with an introduction to essential use cases, discussing their philosophy as developed by Constantine and Lockwood and comparing them to conventional use cases. Then, in section 3, we describe how to write essential use cases, and use role-play to verify that they are correct and consistent. Section 4 shows how essential use cases can be used to design object-oriented systems, reflecting on the role of responsibilities for both essential use cases and object-oriented design, and including a short example. Section 5 then discusses some practical issues such as using essential use cases for systems without human users, development processes, and business process design. We then discuss related work in section 6, and finally present our conclusions.

2 Background

2.1 Use Cases

Jacobson et al. defines a use case in their 1992 book as "a behaviorally related sequence of transactions in a dialogue with the system"[15]. A more recent definition for the Rational Unified Process shows little real change, saying a use case is "a description of a set or sequence of actions, including variants, that a system performs that yields an observable result of value to a particular actor"[14].

The general idea of a use case is to represent intended sequences of interaction between a system (even if not yet implemented) and the world outside that system. This idea is very powerful, for several reasons.

In the early stages of development, use cases help to focus on interactions as a way of eliciting desirable system behavior, and so help capture require-

Example

- A development team is building a desktop version of the game Cluedo. In the Cluedo game each player tries to establish which character is the murderer, in which room the murder was carried out, and with which weapon. In the board game there are cards for all the rooms, weapons and characters and these are distributed among the players except for the three cards that solve the murder (murderer, room and weapon).
- As the game proceeds players take turns and, based on a dice throw, move from room to room. Arriving in a room, they can announce suggestions for that room, the murderer and the weapon. Others players have to show the card if they have it. Players move towards deducing which cards are missing (i.e. have been removed) and so solve the murder. Each time a suggestion is made in a turn, the player can optionally make an accusation which they think solves the mystery and if correct they win, but if incorrect they are eliminated.

Use Case from the Cluedo Game

UC3: Make Suggestion	
Flow of Events	1) Player enters name, weapon, room and character to suggest onscreen 2) The system asks each player in turn if they have the card 3) Each player responds with a Yes/No
Extensions	At 2) Extend to UC4: Make Accusation if player requests

UC3	Make Suggestion		
Main Success Scenario	#	User Intention	System Responses
	1.	Player selects weapon and character.	
	2.		System presents responses from each player up until 1st response
	3.		If no responses given System asks for accusation
	3.		Record Findings
	4.	Optionally, Player accuses.	
	5		System matches accusation with murder envelope – if matched player wins else eliminated

Advanced Software Engineering

Chapter Seven

UML Modelling - Use Cases for Design

Learning Outcomes

Know role use cases for software design

Know how to use Essential Use Cases

Text:
Biddle Paper on Essential Use Cases
Larman Chapter 9 & 31