OBJECT ORIENTED DESIGN

with UML and Java

Part 1

Identifying Actors and Responsibilities

• LECTURE NOTES •

2005

CONTENTS

CHAPTER A. INTRODUCTION

A.1 OBJECT ORIENTED PARADIGM

Object-oriented thinking

- To illustrate the idea of OOP, let us consider a real-world situation in which a person need to send book as a present to his friend, who appears to leave in another city.
- Of course one can purchase and deliver book himself (even though design an algorithm for that), though there exist another, more realistic approach. One will call a bookstore nearby and ask bookseller Joe to deliver the book. One will tell the book name and friend's address, and he can be sure that right book will be delivered.
- In fact, bookseller Joe does not perform delivery herself. He will call another bookstore in the chain in the city of the friend and pass the order to him. The other bookseller will order the book from the publisher, pack the book, and order delivery service to the friend's door.
- This example illustrates the solution of a problem in an object-oriented way. Idea was to find an appropriate ***agent*** (Joe) and to pass to him a ***message*** containing a request (book order). It is in the ***responsibility*** of the agent now to perform the action in regards to that request. In fact, he will uses some ***method*** (a set of operation) to do this. He may use other agents to perform the action as well. We do not need to know the details of his actions. Moreover, this information is usually ***hidden*** from our inspection. Finally the request is satisfied by a sequence of requests from one agent to another.

  So far we have come to the principles of OOP.

Adapted from

- Timothy Budd, Object Oriented Programming, third edition, Addison Wesley, 2002.

A.2 PRINCIPLES OF OOP

Message Passing principle

- Action is initiated in OOP by the transmission of a message to an agent (which is an object) responsible for the action.
- Message contains the request for the action and the additional information (arguments).
- Receiver object accepts responsibility to carry out the action. In response, object will perform some method (a set of operations).

Information Hiding principle

- Client sending the request (message) need not (should not) know the actual means by which the request is honored.

Object Instantiation principle

- We can make certain assumption about booksellers and even broader – merchants, and expect bookseller Joe to fit general pattern (all can sell and deliver some merchandise).
- In this we can define the following principle:

  All objects are instances of a ***class.***

  The method invoked by an object is determined by the class the object belongs.

All objects of a given class use same method (share same code) in response to similar message.

Class Hierarchies-Inheritance principle

· The principle says:

Classes can be organized into hierarchical inheritance structure.

Subclass (also known as ***child class***, or ***derived class***) will inherit attributes from ***superclass*** (also known as ***parent class***)

A ***superclass*** that is a root for the class hierarchy is called ***base class***.

Abstract class has no direct instances but used to create subclasses only.

· According to that principle, we know that our bookseller Joe belongs to a class of booksellers, which belongs to the class of merchants, who are humans, and therefore animals, and finally material objects.

· According to that hierarchy, we know, for example, that book (the product he sells) will cost as some money (the property of merchants), that Joe has name (human property), that he eats and drinks (animal property), and that he has some weight as any material objects do.

· Class hierarchy is usually depicted as a tree diagram with base class in the root, and arrows pointed into the direction from children to parent classes.

Method binding

· Subclass will inherit the properties (including methods) from a superclass. Moreover class can alter (***override***) the inherited behavior. Which method shall then be invoked in response to a given message? The search always starts from the object, which is the receiver of the message. If the class of the object has no method, either of super-classes (classes up in the hierarchy tree) does, the method from a nearest superclass will be used. For example, we do not need to teach Joe to eat simply because he inherits this functionality from the animal class.

Responsibilities

· A fundamental concept in OOP is to describe behavior in terms of responsibilities. The request for action indicates only the desired result, when the agent is free to use any technique that achieves desired objective.

· By discussing a problem in terms of responsibilities we increase the level of abstraction, which permits greater independence between agents, a critical factor in solving complex problems. The entire collection of responsibilities associated with an object is often described by the term ***protocol***.

Responsibility delegation principle

· The most important aspect of OOP is ***responsibility driven design***, which consists of determination and delegation of the responsibilities.

· When we make an object responsible for specific actions, we expect a certain behavior. At the same time we do not interfere with the implementation of the actions performed by the object and give to it a degree of independence.

· From another side, when there exists an object (a class) responsible for something, we do not repeat the action ourselves but use an existing code for it.

- Responsibility delegation principle (which is implicit from ***message passing*** and ***information hiding*** ones) claims:
- ***If there is a task for the client to perform, a first what client shall do is to find some other existing object to do it***.
- This principle not only relives us from taking a complete control of the actions (which makes programming and program understanding more easy), but helps to create independent and reusable components, which can be reused from one project to another.
- In the conventional programming, it is often that some part of the program is doing something to something else, like modifying some record. Thus one portion of code is often tied by control and data connections to many other sections of the system, usually through the use of global variables, pointers or implementation details of other portion of code. A responsibility driven is aimed to eliminate these links to the minimum extent possible.

Adapted from

- Timothy Budd, Object Oriented Programming, third edition, Addison Wesley, 2002.

A.3 OBJECT ORIENTED PROGRAMMING LANGUAGES

- An object-oriented language has objects, classes, inheritance and polymorphism.
- An object oriented program is community of agents working together to address a common purpose.

Objects

- An object is a structure in which both the data (the object's **attributes**) and operations on the object (the object's **methods**) are **encapsulated**. For example a *Date* object (see Figure 1) might store the date as the number of centi-seconds since Januaray 1st, 1900, and would therefore use an integer attribute to store this information. This is convenient for the programmer, allowing dates to be easily subtracted from one another and compared easily. Methods can be provided to allow dates to be created and manipulated in a more human form.

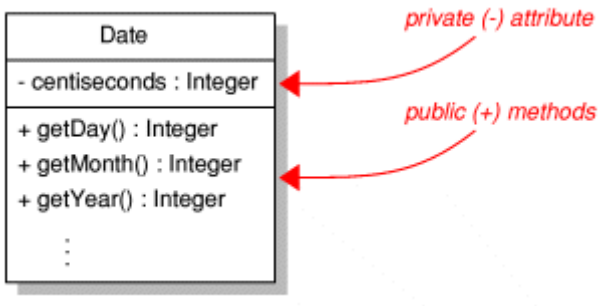

Figure 1: *Date* class.

- Many languages allow access to attributes and methods to be restricted. This allows objects to behave as '**black-boxes**' where access is provided through methods alone and nothing needs to be known about the internal structure of the object. This way a programmer using the *Date* class need not be aware as to how the date is actually stored.

Classes

- Every object is an instance of some class. A class defines the methods and attributes that each instance of the class will possess (intentional view). A class can also be seen as defining the set of objects which are instances of the class (extensional view).

Inheritance

- Inheritance enables new classes to be defined as an extension of another class, inheriting its methods and attributes. Subclasses can provide additional methods and attributes and even **override** methods of the parent class to provide their own behavior.
- We might wish to create a new class which extends our *Date* class but represents a date for a different calendar. We could extend the existing Date class and override some of its existing methods and provide new ones. Polymorphism would allow a instance of this new class to be treated as an instance of *Date*

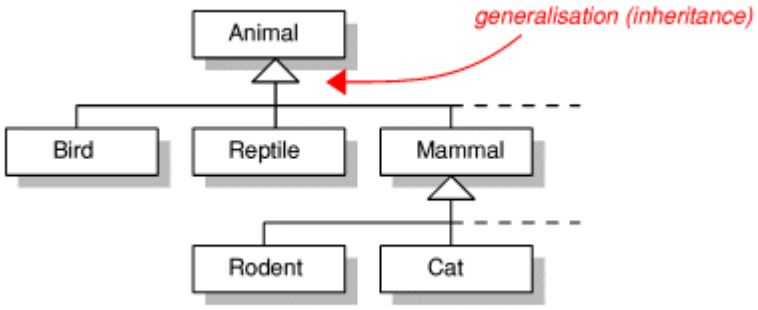

Figure 2: *Animal* class hierarchy.

- Complete class hierarchies can be built up through inheritance, perhaps representing hierarchies in the real world. For example we might have some software which needs to represent *creatures* of some kind (see Figure **2**). Our class hierarchy might then correspond (in part at least) to the biological taxonomy.

Polymorphism

- Polymorphism allows objects of one type (class) to be treated as though they are instances of a more general type (some superclass). So we could treat both *Cat* objects and *Rodent* objects as though they were more general *Mammal* objects since both *Cat* and *Dog* are subclasses of *Mammal*. Likewise a *Cat* object and some *Reptile* object could both be treated as *Animal* objects.
- The reason we can do this is that if one class inherits from another class then that class will have at least the same public methods and attributes as the superclass (i.e. the class which was inherited).

Data-driven programming

- An object-oriented program does not look as a conventional (structural) program, which has pretty strictly defined course of actions – a sequence of action executed one after another.
- Instead OO program is seen as a collection of classes interconnected with each other. How does an object-oriented program work?
- Classes have functionality to create new objects and delegate them some tasks.
- There can be a small procedure like function `main` (C++) or static method `main` (Java) of the application class performing initial actions, i.e. instantiation of a very first object and

requesting it to do some specific action. In some cases even that initial instantiation is taken care solely by the application framework as in the case of Java applets.

- Going further, classes possess specific methods reacting on different situations. These methods are invoked by other objects in the program or by the application framework (e.g. Java Virtual Machine) when an appropriate message is generated by some software or hardware component (e.g. when key is pressed or mouse is dragged over the visual representation of the component).
- The difference between viewing the software in structural terms and from OO prospective can be summarized to well-known quote:
- "Ask not what you can do ***to*** your data structures, but rather ask what your data structures can do ***for*** you".
- To conclude we state that OOP is a ***data-driven programming paradigm***.

Adapted from:

- Thomas Baldwin, Object-Oriented Programming Languages, 1999
  http://www.dcs.shef.ac.uk/~tom/Objects/AutoRDD/ooconcepts.html

A.4 OBJECT ORIENTED DESIGN

OOD

- Working in an object-oriented language is neither necessary nor sufficient condition for doing *object-oriented programming*!
- Major benefit of the OOP occurs when software components and systems are reused from one project to another. The ability to reuse the code implies that the software components have to be organized as independent of other (low *coupling*) and with as meaningful as possible responsibilities (high *cohesion*).
- Ability to create such a reusable software code is not that easily learned. Rather it requires experience, careful examination of case studies, and the use of programming language, in which delegation is supported in a natural and easy to express fashion.
- Whereas OOP considers structure of individual classes and parent/child relationship, OOD considers relationship between groups of classes or objects working together, as well as the process identifying and designing the classes and their relationship. Object Oriented Design (OOD) is concerned with designing the internal logic of a program. It is concerned neither with how the user interface represents that logic nor with how data are stored.
- The input for the design process is the *specification* of the required software system. The end point is a description of the software components: classes, objects and methods and how they interconnect.
- An important distinction between a specification and a design is as follows. A specification says what is required and the design says how it will be accomplished. The specifications are written in the natural language or special formal language (for the formal specifications), and design is expressed in one of several notations (such as CRC cards, UML diagrams, algorithms, and system design documentation).

Unified modeling Language (UML)

- UML (Unified Modeling Language) is a standard notation for the modeling of real-world objects as a first step in developing an object-oriented design methodology. Its notation is derived from and unifies the notations of three object-oriented design and analysis methodologies:

- Grady Booch's methodology for describing a set of objects and their relationships
- James Rumbaugh's Object-Modeling Technique (OMT)
- Ivar Jacobson's approach which includes a use case methodology

Responsibility-Driven Design (RDD)

- Most important aspect of OOP is a design technique driven by determination and delegation of responsibilities, known as *responsibility-driven design (RDD)*. In OOP system every object is responsible for performing some action, and when we make an object responsible for performing the actions, we expect a certain behavior. Responsibilities also imply a degree of independence and noninterference. Unlike procedural programming, that is usually doing something to something else, OOP program works as a collection of agents, issuing directives to each other (due to *responsibility delegation*) and expecting that desired result will be produced without the interference and without even knowing the implementation of the actions (due to *information hiding*).
- Responsibility-driven design aims to generate the classes for a system by considering their responsibilities rather than determining classes by the data they contain (as in data-driven design). A responsibility is a behavioral abstraction - what the class does rather than what it contains.
- The problem with a data-driven approach to software design is that there is a reliance on the way data is stored in a class. Algorithms may relay on this, making it difficult to change the way in which data is stored. Data-driven design breaks encapsulation.
- RDD merges communication paths between classes, thus reducing the coupling between classes, partitions and layers subsystems and generates design patterns (DP).

Design Patterns (DP)

- OOP techniques (composition, inheritance, polymorphism, etc) had finally provided a way to create software from general-purpose interchangeable components. Though it is still up to the programmer to build the software on top of these components, and that is a challenging task for a several reasons:
- OOP techniques themselves provide the means for producing reusable components, but not guidelines for how such a task should be performed.
- The process of producing reusable components is more difficult that making a specialized software, and often the benefits of producing reusable components cannot be often realized within a single project.
- Because many programmers have little formal training, or have not followed the recent programming innovations, they may not be aware of the mechanisms available for the development of reusable software components.
- Every time new problem is encountered the first thing the programmers usually consider to do is to overlook the previously solved problems for a similarity and to use them as a model for the forthcoming solution. This idea is employed in the software *design patterns (DP)*.
- Design pattern describes a proven solution to a problem which can be used as a model for handle many other problems in a similar way. Patterns became important in the development of Object-Oriented programs because they aid in designing architecture and relationship between software components at a higher level of abstraction than classes.
- At the highest level of abstraction the OOP program is seen as a collection of interacting agents. Certain types of relationships appear over and over in many different applications. Design patterns extract the fundamental features of these associations. Design patterns

speed the process of finding a right architecture or design solution for the software without the need to "reinvent the wheel"!

Refactoring

- Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring.

OO Design Strategies

- *Responsibility-Driven Design* - http://www.wirfs-brock.com/
- *Responsibility-Based Modeling* – http://alistair.cockburn.us/crystal/articles/rbm/responsibilitybasedmodeling.html
- *Rational Unified Process* – http://www-306.ibm.com/software/awdtools/rup/
- *The Enterprise Unified Process* – http://www.enterpriseunifiedprocess.info/
- *OMG Model Drive Architecture (MDA)* – http://www.omg.org/mda/
- *Agile MDA* – http://www.agilemodeling.com/essays/mda.htm
- *Agile Model Driven Development (AMMD)* – http://www.agilemodeling.com/
- *Domain Driven Design* – http://www.domaindrivendesign.org/
- *Discovery* – http://www.dcs.shef.ac.uk/~ajhs/discovery/
- *Catalysis* – http://www.catalysis.org/
- *eXtreme Programming* – http://www.extremeprogramming.org/
- *OPEN Process* – http://www.open.org.au/
- *ICONIX Unified Object Modeling* – http://www.iconixsw.com/Spec_Sheets/UnifiedOM.htm
- *Refactoring* – http://www.refactoring.com/

A.5 INTRODUCTION INTO RDD

Why RDD

- In a responsibility-based model, objects play specific roles and occupy well-known positions in the application architecture. It is a smoothly-running community of objects. Each object is accountable for a specific portion of the work. They collaborate in clearly-defined ways, contracting with each other to fulfill the larger goals of the application. By creating such a "community of objects," assigning specific responsibilities to each, we build a collaborative model of our application.
- Objects are more than simple bundles of logic and data ... they are *service-providers*, *information-holders*, *structurers*, *coordinators*, *controllers*, and *interfacers* to the outside world! Each must know and do its part! Thinking in these terms enables you to build powerful, flexible applications. Other design methods tend to focus on logic and data alone. They leave out the big picture and miss the point of objects. Responsibility-Driven Design, on the other hand, offers practical advice for designing, implementing and redesigning responsibilities.

- Responsibility-Driven Design enables you think clearly about object design and to leverage object technology to its full advantage.
- "***Understanding responsibilities is key to good object-oriented design***" said Martin Fowler, noted object expert and author of UML Distilled.
- The shift from thinking about objects as data + algorithms, to thinking about objects as roles + responsibilities can be a profound one. It impacts all development activities. During analysis, we think about what the system is responsible for. We map system responsibilities to appropriate object roles and responsibilities during design. We use responsibilities to define the interfaces of our classes and their test plans during coding. Programmers still reason about and discuss object responsibilities while they cope with myriad coding details. When details are hidden in "helper" code they can be changed and extended swiftly without affecting the implementation of major responsibilities. In a complex world of code and data structures, object responsibilities marry the way we think about our applications to its invisible world of executable code. Quite simply, ***responsibilities are the best way to think about the behaviors of objects***.

Responsibility and noninterference

- Design technique driven by determination and delegation of responsibilities is known as *responsibility-driven design*.
- In OOP system every object is responsible for performing some action, and when we make an object responsible for performing the actions, we expect a certain behavior.
- Responsibilities also imply a degree of independence and noninterference. Unlike procedural programming, that is usually doing something to something else, OOP program works as a collection of agents, issuing directives to each other (due to *responsibility delegation*) and expecting that desired result will be produced without the interference and without even knowing the implementation of the actions (die to *information hiding*).

Programming in the Large

- OOD is better understood and has direct benefit in large scale projects. The difference between development simple application and large scale projects is as follows:
- Programming ***in the Small***:
- Code is developed usually by a single programmer who understands all aspects of a project
- The major problem: the design and development of algorithms
- Programming ***in the Large***:
- The system is developed by a team of programmers, and no one can know a system in total
- The major problem is management of details and communication or interface between diverse system components.
- Most student projects can be though as programming in the small where legacy programming methods can be applied. OOP are best understood in response to the problems encountered while programming in the large.

A.6 COMMON TASKS (STEPS) IN RDD

Identify the behavior of the system.

- The behavior is usually understood long before any other aspect. It is similar to formal specifications with a difference that it can be described in terms meaningful for both the programmer and the client almost from the moment the idea is pictured.

Refine specifications

- In this step the goal is to refine specification by creating the *scenarios* for the application and working trough them. One objective is to get a better "look and feel" of the eventual product. The specification, defining what and how the system does, can be carried back to the client to see if it agrees with the original conception.

Work through scenarios and Identify components

- During this step the software components and their particular actions are identified. A *component* is an abstract entity performing some task and fulfilling some responsibilities. It can be a *function*, *class* or a *pattern* (that is collection of other components). It must have the following characteristics:
  - A component must have a small well-defined set of responsibilities
  - A component must be independent (interact with other components to the minimal extent possible).
- In order to discover components and their responsibilities, the programming team walks through the scenarios acting out the running of the application as if it would be done in a working system. Every activity that must take place is identified and assigned to some component as a responsibility.
  - Often this proceeds as a cycle of *what/who* questions. First the programming team identifies *what* activity need to be performed, which is immediately followed by the question of *who* performs the action.
  - Any activity that is to be performed must be assigned as a responsibility to some component.
- Some decisions concerning the single components can be also postponed until other components are identified or the system functionality is well understood.

Give component a physical representation

- As a part of this process it is often useful to represent a component using small index cards. On the face of the card the programming team writes the name of the software component, the responsibilities of the component, and the names of collaborators: other components with which the component must interact. Such cards are known as component-responsibility-collaborator cards (CRC cards), and are associated with each software component. While working through scenarios it is useful to assign CRC cards do different member of the design team. The member of the team holding the component card acts as the respective component during the scenario simulation. He or she describes the activities of the software system passing control to another member when the software system requires the services of another component.

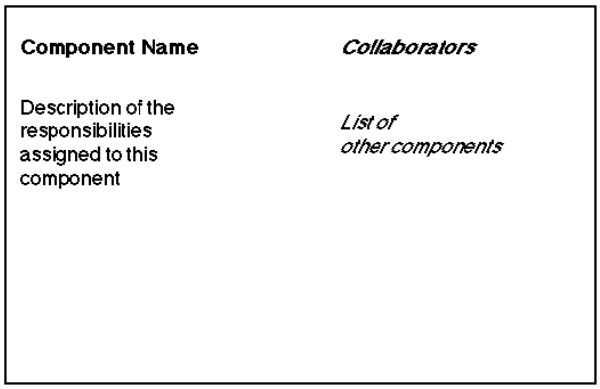

Figure 3. CRC card.

Starting documentation

- At this step the development of documentation shall begin. Two documents should be essential parts of any software system: the *user manual*, and the *system design manual*.
- The *user manual* describes the interaction with the system from the user's prospective. Since the scenarios are closely matching the user's possible behavior, the development of the user manual naturally follows the scenario.
- The *design documentation* records the major changes during software design, and should be produced when these decisions are fresh in mind of the developer, and not after the fact when many of the relevant details will have been forgotten. Too soon the focus will move to the level of individual components or modules, so it is good time to document the system on the general level of hierarchy.
- Arguments for and against any major design alternatives should be recorded, as well as factors that influenced the final decisions. A *log of the project schedule* should be maintained. Both manuals shall be refined with evolution of the software over time.

Formalizing the components

- During this step the components gain their final formal representation. It became clear what they do look like and what function they serve for. The following factors shall be considered during the formalization process.

1. *Behavior and state*

- The components are characterized as a pair consisting of the behavior and state:
- *Behavior* is defined as the set of actions it can perform. The complete description of all the behavior for the component is called *protocol*.
- *State* represents all the information held within component. State is not static and can change over time. Not necessary all components maintain state information.

2. *Coupling and cohesion*

- *Coupling* and *cohesion* are two important design concepts for the components.
- *Coupling* describes the relationship between different components. In general, it is desirable to reduce the coupling as much as possible, since independence of other components gives freedom in development, modification and reuse.
- In practice coupling is increased when one software component must access values that held by another. This could be avoided by moving the task of accessing somebody else's data into list of the responsibilities of the other component itself. For example, the component can be made responsible for drawing itself (e.g. making it visual component) instead of having another component to draw it according to its data representations.
- *Cohesion* is the degree to which the responsibilities of a single component form a meaningful unit. High cohesion is achieved by associating in a single component tasks that are related in some manner (for example correlated through the need to access common data area).

3. *Information hiding (interface and implementation)*

- The component shall hide the behavior showing only how the component can be used by either user (other component) or future developer and not the detailed actions it performs. The *interface* and *implementation* shall be (and this is extremely important in large projects!) separated accordingly to the *Parnas's* principles:
- ***The developer of the component must provide the intended user with all the information necessary to make effective use of the services provided by the component and should provide no other information***.

- ***A class definition must provide the user with the information necessary to manipulate the instances of the class and <u>nothing more.</u>***

- ***The developer of the component must be provided with all the information necessary to carry out the given responsibilities and should provide no other information.***
  - ***A method must be provided with all the information necessary to carry out its given responsibilities, and <u>nothing more</u>.***
- This principle divides an object into to spheres (separates the notion of ***what*** is to be done from ***how*** it is to be done).
  - ***Interface*** – an object as it is seen to the user. It consists of all declarations necessary to manipulate the object: data types, member functions prototypes, functionality to access member variables, and sometimes publicly available data. It is also known as a ***contract*** between class designer and the user of the class, and
  - ***Implementation*** – a view from within the object, which is hidden to the user. It consists of all member variable declaration, and member function implementations, which also takes control over the member variables

4. *Postponing decisions and Preparing for changes:*

- Don't make important decisions early and became "design-fixated".
- No matter how carefully one tries to develop the initial specifications and design of a software system, it is almost inevitable that the changes in the user's needs or requirements will force the changes in the software. Programmers shall prepare for such changes and design the system accordingly:
  - The changes shall affect as few components as possible.
  - Most likely sources of change must be predicted, and common interface are developed to isolate from implementation specific details.
  - Dependencies (coupling) between software components shall be reduced.
  - Dependencies of hardware (or software platforms) shall be isolated.
  - Design documentation shall maintain careful records of the design process.

5. *Formalizing the interface.*

- Finally the decision shall be made on the general structure that will be used to implement the component. A component with only behavior and internal state may be made as a function (in Java as a static method; all methods working on similar data types can be grouped in a single utilitarian class). The components with many tasks can be implemented as classes.

6. *Name associations.*

- Names shall be associated with the actions the component can perform. The selection of useful names is extremely important, as names create vocabulary with which the eventual design will be formulated. Names shall be consistent, meaningful, pronounceable, preferably short and evocative (suggestive) in the context of the problem.

The results of this step (formalization) can be expressed in *diagrams* (*UML diagrams*):

- The relationship as well as organization of the components can be well defined using specialized graphical notations. Diagrams can be drawn on various stages of the design process from early time *case studies* and *activity diagrams*, to design stage diagrams such as:

- *Static diagrams* that visualize static relationship between classes such as *ISA* (inheritance), *HASA* (reference) and *USES* (method invocation) associations in class diagrams, and *data flows* in *data-flow diagrams*.
- *Interaction diagrams* (such as sequence diagrams) illustrating course of actions and activities and communications between components during carrying out some task.

Designing the representations

- At this point the design team cab be divided into groups, each responsible for one or more software components. The task now is to transform the description of the component into a software system implementation. This process must start with the designing the *data structures* that will be used by each subsystem to maintain the state information.
- Once data structures are chosen, the descriptions of behavior must be transformed into *algorithms*.

Implementing the components

- Once the design of each software component is laid out, the next step is to *implement* each component's behavior using a particular programming language.
- During this process, it may happen that certain information or action can be assigned to secondary components working "behind the scene". Such components aiding in completion of regular tasks are known as *facilitators*.
- All the necessary *preconditions* a component requires to complete a task successfully, must be properly *documented* as well *verified* on the correctness.

Integration of components

- Once software sub-systems have been individually designed and tested (*unit testing*), they can be integrated into the final product. This is often not a single step but a part of a larger process. Starting from a simple base, components are slowly added to the system and tested.
- *Stubs* (routines with no or limited behavior) are used to replace temporarily missing components. This process is known as *integration testing*.
- An application is finally complete when all stubs have been replaced with working components. The ability to test components in isolation is facilitated by the goal of designing the independent components.

Maintenance and Evolution

- Software *maintenance* describes the activities after the system has been implemented and deployed. Such activities may be facilitated by:
  - Errors (bugs) requiring correction (patching)
  - Changing in the requirements (e.g. regulations)
  - Changing in the hardware,
  - Changing in user expectations,
  - Aim at better documentation.

Adapted from

- R Wirfs Brock and L Wiener. "Responsibility-driven design: a responsibility driven approach", Proc. 4th ACM Conf. Object-Oriented Prog. Sys., Lan. and Appl., pub. Sigplan Notices, 24(10) (1989), 71-76.
- Timothy Budd, Understanding Object-Oriented Programming with Java, Addison Wesley, 2000.

A.7 SIMULATION AND MODELING

Introduction

- We know that an object-oriented program consists of a collection of objects interacting with each other by the mean of messaging passing. We know also that the fundamental problem of object-oriented design is in identifying the objects. Often the task of the program is to simulate real world situations. For example, when developing office-work automation system we simulate users, shared documents, their files, archives and workflows. In a factory automation system we simulate different machinery, queues of work, orders and deliveries. Our goal is therefore to identify the objects in the problem and to model them as objects in the program.

Model-View-Controller architecture

- Abstraction plays a role in the process of modeling. We need only model sufficient information for the problem to be solved, and we can ignore any irrelevant details. If we are creating a personnel record system, we would probably model names, addresses and job descriptions, but not hobbies and preferred music styles.
- The *Model-View-Controller* architecture is a special design pattern created to model the real-world applications. It consists of the three:
  - The *model* that simulates an object from a real world to the give degree of details
  - The *view* that represents the model for the user.
  - The *controller* that controls the model parameters (is the response of the use back to the model).
- The model is usually invisible (except program code). The view is visible on the screen as a graphical image, graphs, diagrams, etc. The controls are visible as GUI elements including scrollbars, buttons, etc. The model is frequently changed by the controller (for example when user interacts with the controller by the mean of its GUI). As a consequence, this action initiates the changes in the view. The model itself exists independently of the view and controller that can be varied from the application to application.
- For example, when modeling the car engine, the model is the engine, the view is the tachometer and the control is the gas pedal.

A.8 CASE STUDY: "THE BALLOON GAME"

- In the following example we briefly outline an application that models the balloon. The objective of the application is to demonstrate the object-oriented design using model-view-controller architecture.

The task

- Our task is to develop an application that models the balloon. The balloon can have different size and location, both can be controlled by the user by the mean of application GUI. The balloon (given its size and location) is displayed as a red circle on the blue background. The application must look similar to the following:

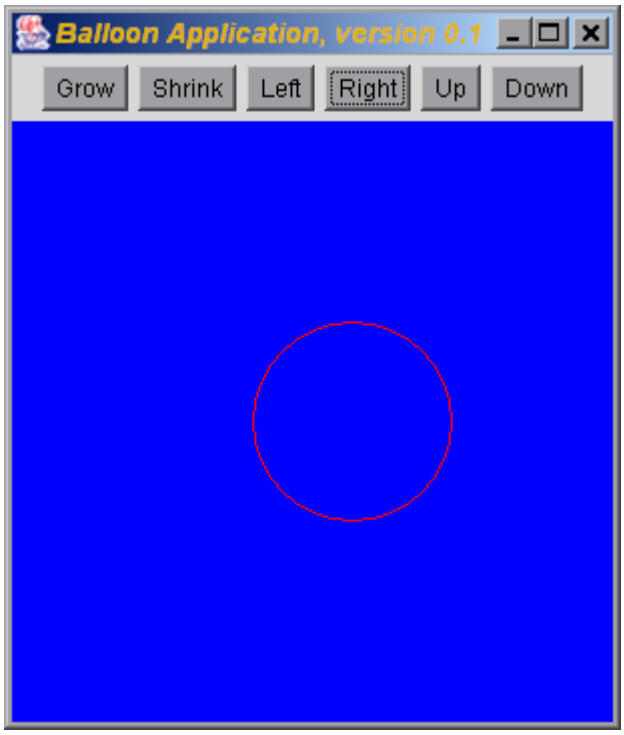

Figure 4. Outline of a user interface.

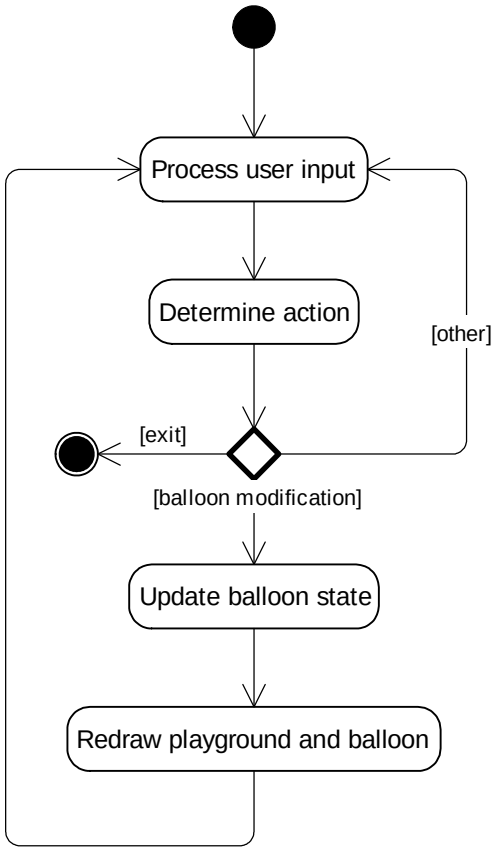

Figure 5. Game scenario.

The scenario

The scenario for the application can be expressed using the following block-diagram.

The cycle starts from processing the user input. Application determines the action, after which it updates balloon size or location, and redraws the playground and the balloon on the screen as shown in the following *block-diagram*:

The component identification

The application (namely **`PlayBalloon`**) will be composed upon the balloon model, its view and its controller. These components are dictated to us by the design architecture we have chosen.

Balloon is modeled as an object that has position in a virtual world and size. The balloon is viewed as a red circle on a blue playground.

The view for the balloon will be an independent from the model visual component (namely **`BalloonView`**) that is displayed in the application window.

The controller for the balloon (**`BalloonController`**) will be able to change the position and the size of the balloon. The controller will have own GUI (implemented as a separate class **`BalloonControlPanel`**) shown as six buttons: four to alter balloon location in respective direction and two to alter the balloon size.

This architecture can be illustrated using the following *UML class diagram*:

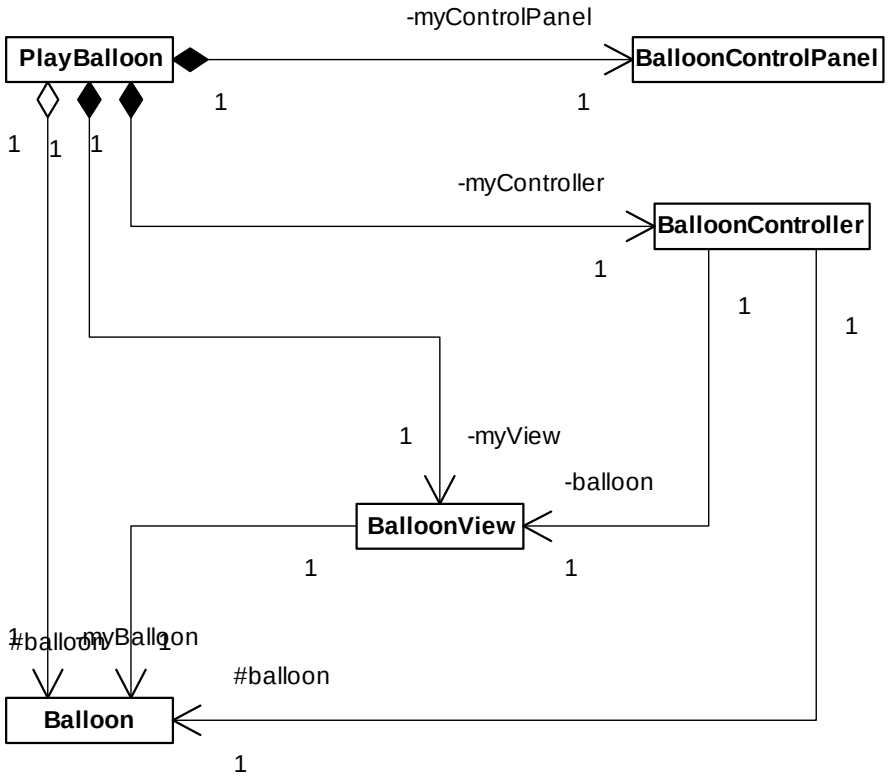

Figure 6. Components of the game.

Considering the above components, the scenario can be illustrated using the *UML activity diagram* with *swimming lanes*. The swimming lanes illustrate what actions are in the responsibility of which component.

Rounded boxes in the diagram represents the activities, and rectangular boxes represent the parameters that result from the previous activity and must be used as an argument for the successive activity

Note: in this diagram, the arrows do not mean the invocation calls; rather they mean the course of action and the order of their execution (and nothing else).

The component design

The following two diagrams illustrate the design of the software components.

First diagram is a UML *class diagram* representing static relationship such as

- *inheritance* – a class extending another class – solid arrow with triangle-head, arrow drawn in direction to super-class
- *aggregation* – object of this class owns one/several/many objects of associated class – represented as solid arrows with diamond (which can be further classified to either *composite* aggregation (solid diamond) or *shared* aggregation (empty diamond). Composite aggregation occurs when associated class makes sense ONLY in the context of the referencing class and makes no sense on its won (e.g. event processing adapter class or inner class, or role in a club);
- other *associations* – object references another object, which is necessary for method invocations – represented as normal arrows; arrow is draw in direction of navigability of the association (can be bidirectional as well)

Another diagram is a UML *sequence diagram* representing object lifetime and communication between objects. Diagram depicts object creation, synchronous and asynchronous message passing (method invocation operations). Asynchronous method

invocation result for example from event generation or repaint requests and is depicted as arrows with half-winged arrowhead.

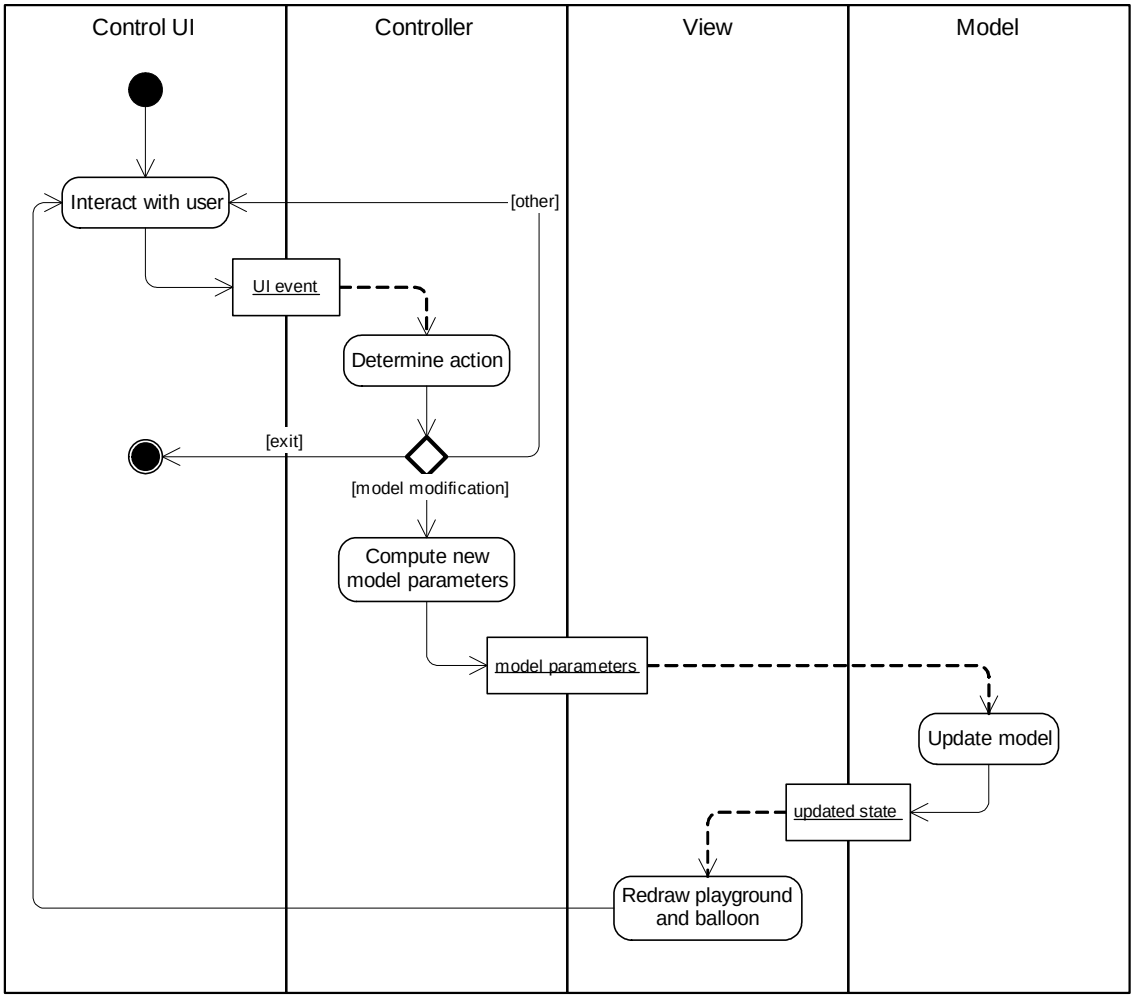

Figure 7. Activity diagram for the game.

In order to minimize coupling between components we perform the following steps:

- We further split the balloon model into two classes: an abstract **`BalloonModel`** and its implementation as **`SimpleBalloon`** (in order to make controller and view independent on particular balloon implementation).
- We let application to have own independent GUI for the controller (**`BalloonControlPanel`**) in order to ease further changes for it.
- We make **`BalloonControlPanel`** independent from the **`BalloonController`** class by referencing it to an interface **`BalloonActionListener`** that defines action commands and **`actionPerformed`** method (via **`ActionListener`** interface). The controller implements this interface in order receive action commands from the GUI.

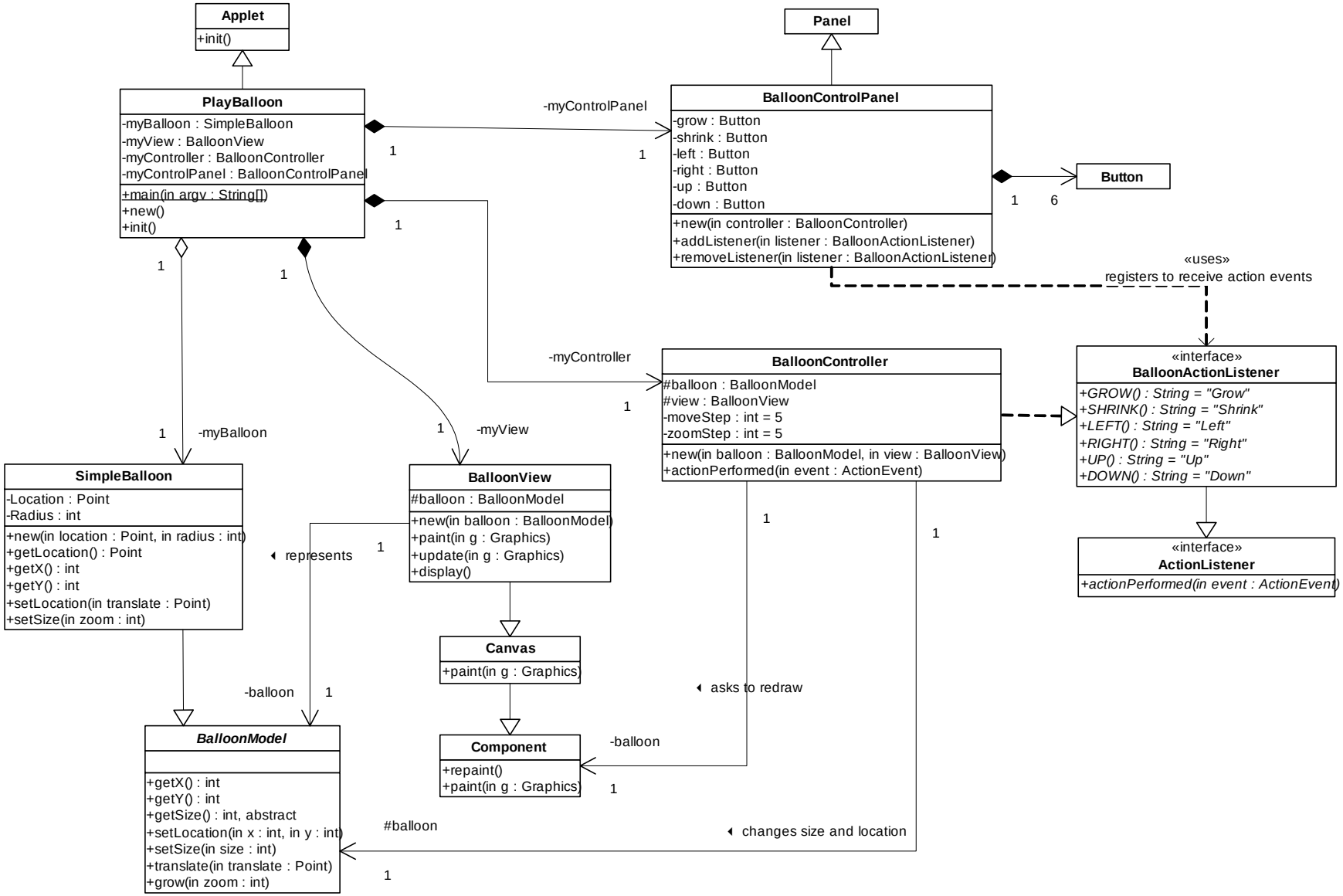

Figure 8. Class diagram for the game.

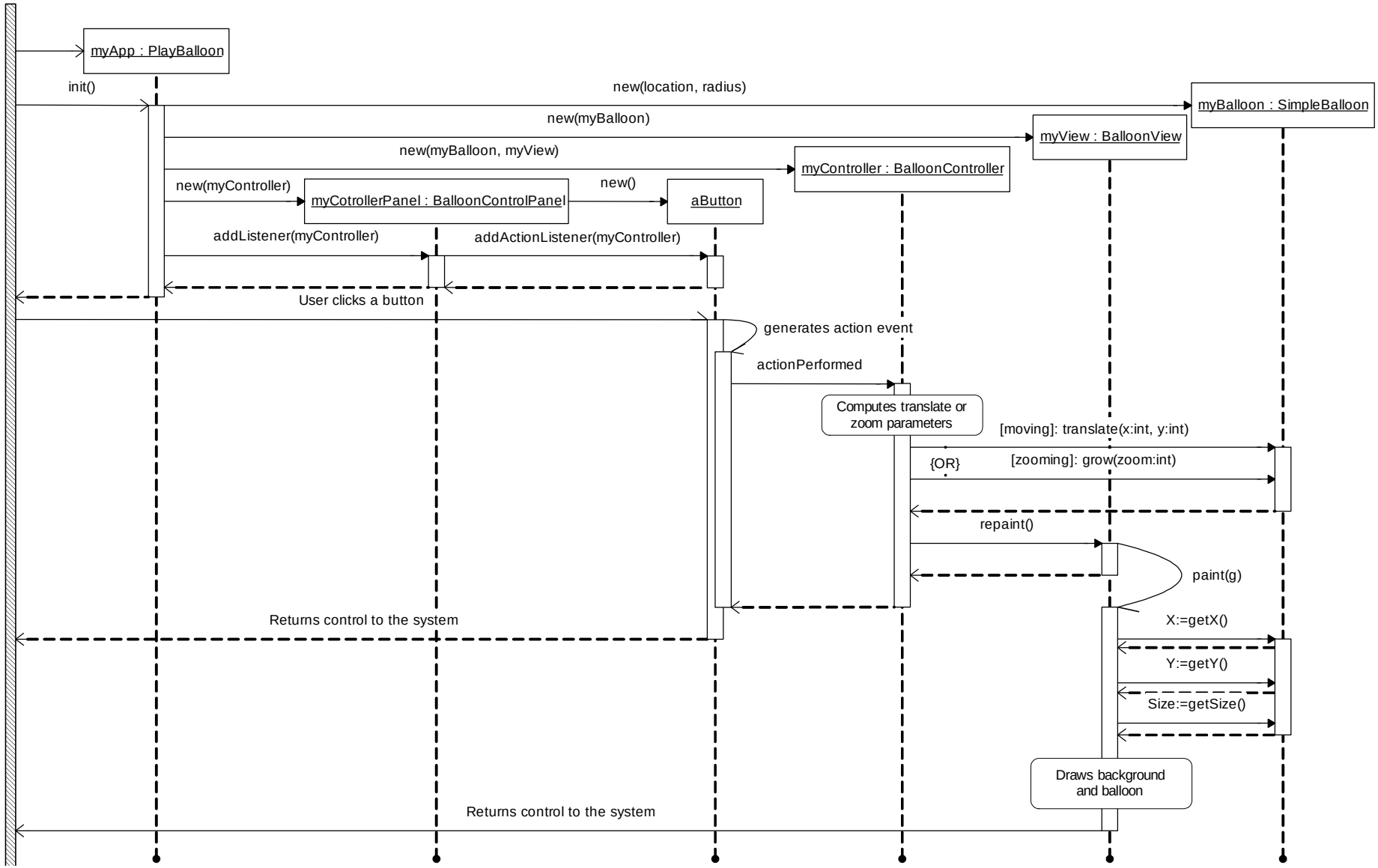

Figure 9. Sequence diagram for the game.

References

- Timothy Budd, Object Oriented Programming, third edition, Addison Wesley, 2002.
- E Gamma, R Helm, R Johnson and J Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software (1995), Addison-Wesley.
- Martin Fowler, Refactoring, http://www.refactoring.com/
- Rebecca Wirfs-Brock and Alan McKean, Object Design: Roles, Responsibilities, and Collaborations, Addison-Wesley, 2002.

Appendix: Java source code for the Balloon game.

The following shows the source code in Java language.

The source code is supplied with the documentation (javadoc) comments.

1. *The class* ***`BalloonModel`*** *– abstract class for the balloon models*

```
import java.awt.Rectangle;
/**
 * Balloon - model for the balloon
 * it has size and location as well as utilitary methods
 * to change location and size in an imaginable world
 *
 * @author Eugene Ageenko
 * @version 0.1 (Apr 9, 2003)
 */
public abstract class BalloonModel {
    /**
     * Moves balloon to a new location.
     * @param x new horizontal coordinate of the balloon
     * @param y new vertical coordinate of the balloon
     */
    abstract public void setLocation(int x, int y);

    /**
     * Changes the size of the balloon
     * @param r new radius of the balloon
     */
    abstract public void setSize(int r);

    /**
     * Returns balloon location.
     * @return horizontal location
     */
    abstract public int getX();

    /**
     * Returns balloon location.
     * @return vertical location
     */
    abstract public int getY();

    /**
     * Returns balloon size.
     * @return size
     */
    abstract public int getSize();

    /**
```

```
     * Moves balloon relative current location
     */
    public void translate(int x, int y) {
        setLocation(getX() + x, getY() + y);
    }

    /**
     * Grows/shrinks the balloon.
     * @param r new radius increment (negative for shrinking)
     */
    public void grow(int r) {
        setSize(getSize() + r);
    }

    /**
     * Returns the bounding #Rectangle of this balloon.
     */
    public Rectangle getBounds() {
        return new Rectangle(getX()-getSize(), getY()-getSize(),
                             getSize()*2, getSize()*2);
    }

    /**
     * Tests if a specified point is inside the boundary of this
     * Balloon
     * @param x,y the coordinates to test
     * @return <code>true</code> if the specified point
     * is contained in the balloon (assumes balloon is a circle);
     * <code>false</code> otherwise.
     */
    public boolean contains(int x, int y) {
        if (getSize() <= 0)
            return false;
        else
            return ((getX() - x) * (getX() - x) +
                    (getY() - y) * (getY() - y))
                   < (getSize() * getSize());
    }
}
```

7. *The class* ***SimpleBalloon*** *– implements balloon model with integral size and location*

```
/**
 * SimpleBalloon - model for the balloon that
 * has integral size and location
 *
 * @author Eugene Ageenko
 * @version 0.1 (Apr 9, 2003)
 */
public class SimpleBalloon extends BalloonModel {

    /** Ballon has radius and central point */
    private int radius;
    private int xCoord;
```

```
    private int yCoord;

    /**
     * Deafult constructor.
     * Creates balloon with default parameters: (5,5,5)
     */
    SimpleBalloon() {
        this(5,5,5);
    }

    /**
     * Parameterized constructor for class balloon
     * @param radius balloon radius
     * @param x new horizontal coordinate of the balloon
     * @param y new vertical coordinate of the balloon
     */
    SimpleBalloon(int radius, int x, int y) {
        this.radius = radius;
        this.xCoord = x;
        this.yCoord = y;
    }

    /**
     * Moves balloon to a new location.
     * @param x new horizontal coordinate of the balloon
     * @param y new vertical coordinate of the balloon
     */
    public void setLocation(int x, int y) {
        xCoord = x;
        yCoord = y;
    }

    /**
     * Changes the size of the balloon
     * @param r new radius of the balloon
     */
    public void setSize(int r) {
        if (r>0) radius = r;
        else radius = 0;
    }

    /**
     * Returns balloon size.
     * @return size
     */
    public int getSize() { return radius; }

    /**
     * Returns balloon location.
     * @return horizontal location
     */
    public int getX() { return xCoord; }

    /**
```

```
     * Returns balloon location.
     * @return vertical location
     */
    public int getY() { return yCoord; }

}
```

8. *The class* ***BalloonView*** *– the View.*

```
import java.awt.*;
/**
 * A view of the Balloon in an imaginable world (playground)
 * Implemented as blue canvas (Panel) with red circle
 * representing the balloon
 *
 * @author Eugene Ageenko
 * @version 0.1 (Apr 9, 2003)
 */
public class BalloonView extends Canvas {

    private Dimension preferredSize;
    // reference to the balloon model
    protected BalloonModel balloon;

    /** A constructor for the view
     *  @param b reference to the Balloon
     */
    public BalloonView (BalloonModel b) {
        this.balloon = b;
    }

    /**
     * Sets preferred size for the view.
     * @param d preferred size referenced as Dimension object.
     */
    public void setPreferredSize(Dimension d) {
        if (preferredSize == null)
            preferredSize = new Dimension(d);
        else
            preferredSize.setSize(d);
    }

    /**
     * Returns preferred size for the view. Use by layout manager.
     * @return preferred size referenced as Dimension object.
     */
    public Dimension getPreferredSize() { return preferredSize; }

    /**
     * Move balloon to the center of the playground.
     */
    public void centerBalloon() {
        Dimension dim = getSize();
        balloon.setLocation(dim.width/2, dim.height/2);
```

```
        repaint();
    }

    /** Returns a reference to a balloon */
    public final BalloonModel getBalloon() { return balloon; }

    /**
     * Draw playground and balloon.
     */
    public void paint(Graphics g) {

        // Canvas automatically fills with the current background
        // g.clearRect(0, 0, getSize().width, getSize().height);

        // Draw balloon
        g.setColor(Color.red);
        int x = balloon.getX();
        int y = balloon.getY();
        int r = balloon.getSize();
        g.drawOval(x - r, y - r, r*2-1, r*2-1);
        // g.fillOval(x - r, y - r, r*2, r*2);
    }
}
```

9. *The class* ***`BalloonControlPanel`*** *– the GUI for the controller*

```
import java.awt.*;
import java.awt.event.ActionEvent;
/**
 * BalloonControl is a Control Panel with GUI to control the Balloon
 * @author Eugene Ageenko
 * @version 0.1 (Apr 9, 2003)
 */
public class BalloonControlPanel extends Panel {

    private Button grow, shrink, left, right, up, down;

    /**
     * The constructor.
     * @param controller the listener for the events generated by
     * the control panel
     */
    public BalloonControlPanel(BalloonActionListener controller) {
        grow = new Button("Grow");
        grow.setActionCommand(controller.GROW);
        grow.addActionListener(controller);
        add(grow);

        shrink = new Button("Shrink");
        shrink.setActionCommand(controller.SHRINK);
        shrink.addActionListener(controller);
        add(shrink);

        left = new Button("Left");
```

```
        left.setActionCommand(controller.LEFT);
        left.addActionListener(controller);
        add(left);

        right = new Button("Right");
        right.setActionCommand(controller.RIGHT);
        right.addActionListener(controller);
        add(right);

        up = new Button("Up");
        up.setActionCommand(controller.UP);
        up.addActionListener(controller);
        add(up);

        down = new Button("Down");
        down.setActionCommand(controller.DOWN);
        down.addActionListener(controller);
        add(down);
    }

    /**
     * Registers new controller as a listener to receive Balloon specific
     * events (action events on the moment) from the control panel.
     * @param controller the listener to add
     */
    public void addListener(BalloonActionListener controller) {
        grow.addActionListener(controller);
        shrink.addActionListener(controller);
        left.addActionListener(controller);
        right.addActionListener(controller);
        up.addActionListener(controller);
        down.addActionListener(controller);
    }

    /**
     * Removes the specified listener.
     * @param controller the listener to remove.
     */
    public void removeListener(BalloonActionListener controller) {
        grow.removeActionListener(controller);
        shrink.removeActionListener(controller);
        left.removeActionListener(controller);
        right.removeActionListener(controller);
        up.removeActionListener(controller);
        down.removeActionListener(controller);
    }
}
```

10. The interface ***`BalloonActionListener`*** *– defines*

```
import java.awt.event.ActionListener;
/**
 * Interface for the class that will listen to events generated by
 * the BalloonControlPanel
```

```
 *
 * @author Ageenko
 * @version 0.1 (Apr 9, 2003)
 */
public interface BalloonActionListener extends ActionListener {
    // constants for the commands that listener shall process
    public static final String GROW   = "grow";
    public static final String SHRINK = "shrink";
    public static final String LEFT   = "left";
    public static final String RIGHT  = "right";
    public static final String UP     = "up";
    public static final String DOWN   = "down";
}
```

11. The class ***BalloonController*** *– the Controller.*

```
import java.awt.event.ActionEvent;
import java.awt.*;
/**
 * BalloonControl is a control for Balloon.
 * It can handle events from controlling components from the
 * Control Panel.
 * It can change the state of the associated balloon accordingly,
 * and finally it asks associated view to update itself trough the
 * call to repaint() method.
 *
 * @author Eugene Ageenko
 * @version 0.1 (Apr 9, 2003)
 */
public class BalloonController implements BalloonActionListener {

    private int mstep, zstep;
    private BalloonModel balloon;
    final private Component view;

    /**
     * The constructor.
     * @param balloon balloon model to control
     * @param view component representing the view
     */
    public BalloonController(BalloonModel balloon, Component view)
    {
        this(balloon,view,1,1);
    }

    /**
     * The advanced constructor.
     * @param balloon balloon model to control
     * @param view component representing the view
     * @param mstep step for moving when moving button is pressed
     * @param zstep step for zooming when zooming button is pressed
     */
    public BalloonController(BalloonModel balloon, Component view,
                             int mstep, int zstep)
```

```
    {
        this.balloon = balloon;
        this.view = view;
        this.mstep = mstep;
        this.zstep = zstep;
    }

    /**
     * Event handler. Processes events with action commands specified in
     * BalloonActionListener interface
     * @see BalloonActionListener
     */
    public void actionPerformed(ActionEvent event) {
        // System.out.println("Interaction with user");
        if (GROW.equals(event.getActionCommand()))
            balloon.grow(zstep);
        if (SHRINK.equals(event.getActionCommand()))
            balloon.grow(-zstep);
        if (LEFT.equals(event.getActionCommand()))
            balloon.translate(-mstep,0);
        if (RIGHT.equals(event.getActionCommand()))
            balloon.translate(mstep,0);
        if (UP.equals(event.getActionCommand()))
            balloon.translate(0,-mstep);
        if (DOWN.equals(event.getActionCommand()))
            balloon.translate(0,mstep);

        // schedules view for the repaint
        if (view!=null) view.repaint();
    }
}
```

12. *The class* ***PlayBalloon*** *– the application (can work ass applet though).*

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
/**
 * Description: Demo application to outline principles of OO Design
 * Copyright: Copyright (c) 2002 by author
 *
 * @author Eugene Ageenko
 * @version 0.1
 */
public class PlayBalloon extends Applet {

    private SimpleBalloon myBalloon;
    private BalloonView myView;
    private BalloonController myController;
    private BalloonControlPanel myControlPanel;
    private final int step = 5;
    private boolean firsttime = true;
```

```
    public static void main(String[] args) {
        System.out.println("Starting...");

        PlayBalloon app = new PlayBalloon();
        app.init();

        // creating a window to put int the PlayBallon
        //  applet if running as application
        BalloonFrame frame =
          new BalloonFrame("Balloon Application, Simple version");
        frame.add("Center",app);
        // set absolute size for the frame window
        // frame.setSize(400,300);
        frame.pack(); // let frame window determine its size
        app.myView.centerBalloon(); // center balloon in the window
        frame.show();
    }

    /**
     * Nested class that defines a frame for the applet allowing
     * to run it as application. The cass is defined static because its
     * instance not need to be the part of the enclosing class instance.
     * This class makes sence only for our Applet class.
     */
    static class BalloonFrame extends Frame {

        /**
         * WindowAdapter is an abstract class that implements
         * WindowListener interface. We use WindowAdapter to process
         * window events so that we do not need to
         * implement every method defined in WindowListener interface.
         */
        class MyWindowAdapter extends WindowAdapter {
            public void windowClosing(WindowEvent e) {
                System.out.println("Closing...");
                setVisible(false);
                dispose();
                System.exit(0);
            }
            public void windowIconified(WindowEvent e) {
                System.out.println("Minimized...");
            }
            public void windowDeiconified(WindowEvent e) {
                System.out.println("Restored...");
            }
        }

        public BalloonFrame (String name) {
            setTitle(name);
            addWindowListener(new MyWindowAdapter());
        }

    }
```

```
    /**
     * Initializes the applet.
     * The method creates instances of the Model, View and Controller
     * and adds visual components to the layout.
     */
    public void PlayBalloon() {
        System.out.println("Initialized...");
        setLayout(new BorderLayout());

        myBalloon = new SimpleBalloon();
        myView = new BalloonView(myBalloon);

        myView.setPreferredSize(new Dimension(300,300));
        myView.setBackground(Color.blue);
        myView.setForeground(Color.black);
        add(myView, BorderLayout.CENTER);

        myController = new BalloonController(myBalloon, myView,
                                             step, step);

        myControlPanel = new BalloonControlPanel(myController);
        add("North",myControlPanel);
    }

    /**
     * Place balloon into initial position when applet is started
     */
    public void start() {
        // center balloon when applet is started
        if (firsttime) {
            myView.centerBalloon();
            firsttime = false;
        }
    }
}
```

CHAPTER B. RESPONSIBILITY-BASED MODELING

B.1 INTRODUCTION

Foreword

- Responsibilities are a way to state the rationale of the system design. The identification and allocation of responsibilities across the system are the primary activity of design of business models and software. The identification and allocation of responsibilities as a primary activity is followed closely and accompanied by reuse of existing components. In object-oriented modeling and design, responsibilities are defined and allocated at the same time, whereas in other techniques, such as structured analysis, they are defined but not allocated.
- People seem naturally well equipped to work with responsibilities and their allocation; perhaps just from the way our societies are built. Dr. J. Fisher, professor emeritus of Towson State University, wrote, "*A rational society, be it a corporation or a country, can only maintain itself if personal responsibility and accountability are at its core; that is, from top to bottom, every agent or citizen must be empowered to conduct her or his role and to be fully accountable for its performance.*"
- ***Responsibility-based modeling*** (RBM), as described here, is essentially the same as ***Responsibility-driven design*** (RDD), as described by Rebecca Wirfs-Brock and used in Beck and Cunningham's CRC card technique. In RDD, emphasis is placed on inventing software classes. Responsibility-based modeling is appropriate for more than designing software classes. It can be applied equally well to the partitioning of a system into subsystems. A subsystem is just a one-of-a-kind object, perhaps the only instance of its type. Responsibility-based modeling appropriately defers concern about the internal structure of the subsystem and focuses on its role and interaction with its colleagues.

RBM product

- ***The product*** of RBM design process consist of
  1. Components from which the system is constructed.
  2. Responsibilities and services provided by them.
  3. The way they satisfy the requirements as stated in the use cases.
- The team is responsible for researching and eventually knowing of the material that can be made available to the design. That includes any business or data models that exist, documents stating business rules, design patterns, frameworks, and program components. The business or data models supply candidate names for components, and specify cardinality relationships. The business rules supply information about collaborations and likely areas of change. Design patterns provide ideas from previous designs. The frameworks and program components supply ready-make artifacts that can reduce the new work that must be done. It is for the partitioning team to decide which new components to create and introduce into the system, and to simplify the design or design task wherever possible.
- Responsibility-based modeling is a recursive technique. It is likely that the team will create a set of components that need further partitioning. Those components may be partitioned by the same or another team. Each team is responsible for the quality of the way their components work together, and the way they simplify, ease or protect the design.
- It is not necessary that beginning designers will *invent* proper placement from the start, but rather, that when presented with two designs, they will be able to appreciate the

improvement of one over the other. Such ability gives a person a chance to improve their design skills over time.

- The responsibilities act as requirements on the subcomponents. A principle of the technique is that the designers are responsible for determining that they exist or can be built. The designers say, in effect, "*If we had these components, with these capabilities, we could deliver the function. We certify that such components either exist or can be built*."

B.2 THE CORE OF RBM

Five core activities

- This technique uses five activities: ***preparation, invention, evaluation, consolidation, documentation***.
- In ***preparation***, the requirements to be used in the design session are collected. By the beginning of the session, the team has decided what portion of the system is being designed (limiting the breadth of the design activity), at what level the design is addressed (limiting the depth of the design activity), and what use cases are needed to address the design of the system at that breadth and depth.
- In ***invention***, the object types are posed, and components are freely named as it appears they may be useful in carrying out the scenario. Responsibilities are assigned, provisionally. Names get changed. More components get named than eventually get used. Sometimes components from other levels get named and used until the level difference is discovered and they get put on the side for future use.
- In ***evaluation***, a series of questions and scenarios are posed, to stress test the design. The posing of questions checks the validity, naming and long-term usefulness of the component. The posing of alternate scenarios is "variation analysis", sometimes called "robustness evaluation". The assumptions of the requirements and the implementation technology are varied to see how much or how little of the design must change to accommodate them. A design is considered better if the changes required can be localized to fewer components.
- In ***consolidation***, the components that have survived the first two activities are collected and reexamined for their names and their level. Components from a lower level are noted and put to the side for later use. Names are checked for meaningfulness, stability and mnemonic value.
- In ***documentation***, the reasons why a particular division of responsibilities was created are written down, along with scenarios that illustrate the use of the division of responsibilities. Interaction diagrams for those key scenarios are drawn, using only the components of the level being designed. The components that already exist are identified; the components needing to be designed are specified.
- The design uses those activities roughly in sequence. All five activities must be used before the design is considered complete. It does happen that the activities are used out of sequence also. That is, often during component invention, new scenarios are invented; variation analysis is done to make a selection between two design choices. Consolidation may be done whenever someone notices that a set of components are at a different level. The reason for a particular allocation of responsibilities may be documented whenever the group feels it necessary, so as not to forget the reason. Some people like to document the interactions immediately; others prefer to wait until the design has stabilized. In all cases, consolidation and documentation have to be checked at the end, both for completeness sake and to make sure the team is in agreement.

Terms used:

system:

- the system under design. This term could refer to the entire application or major deliverable; it could refer to one of the components created from a decomposition of the major deliverable; it could refer to a subsystem. It could refer to an organization of people, computer hardware, or software, or a combination. Whatever it is that is being designed. The only requirement is that the system be composed of communicating parts, because responsibility-driven partitioning works with the messages and information sent between the parts of the system.

component:

- Whatever the system is composed of that communicates with other parts of the system. A component could be a system in its own right, an organization of people, a person, computer hardware or software. It could be a type, class or object in an object-oriented system. It could be a purchased vendor package that must be integrated into the rest of the system. It could be a set of operating system services or a database. It could be an program that will not be designed using object-oriented techniques. In this text, *component* is used as in the sentences: a system is partitioned into components; this technique shows how to specify the components that make up the system

responsibility:

- a promised set of services; the role of the component in a system. Responsibilities are a component's contribution to a system, as they are the services the designers and users of the system rely upon the component to carry out. The component's role summarizes the services it provides in the context of the system.

capability:

- the possibility of providing a set of services; a responsibility taken out of context. In the context of a functioning system, each component has a responsibility toward carrying out the complete function. When the component is put into the component library, those responsibilities are taken out of context. To the next designer, they look like the capabilities of the component. The next designer will consider how those capabilities can act as responsibilities in the context of the system under design. In this book, the word "responsibilities" is used wherever possible, for consistency and simplicity. The word capabilities is used when referring to components in the library, out of the context of a particular design.

B.3 DISCUSSION OF RESPONSIBILITIES

Types of responsibilities

- There are two kinds of ***responsibilities:***
- ***active responsibility -***
  - the responsibility to *do* something,
- ***contact point responsibility -***
  - the responsibility *to act as a contact point for information*, in effect, mediating the information.
- In the first case, the responsibility is described using an active form of verb and the meaning of the responsibility is quite clear. The second needs further explanation.
- It often happens that a component is responsible for providing information to other components and staying current on that information. It is a valid "contact point" for the information, and its responsibility is to remain a valid contact point for the information,

however the information evolves, and however the designers eventually decide to store or compute it.

- There are people who come to the design session thinking about the "state data" that a component keeps, or the "attributes" an entity has. Neither of these is appropriate for responsibility-based modeling, because it is concerned only with the services that components provide each other. The equivalent consideration to attribute and state data is being a contact point.
- There is no external difference between active and contact point responsibilities. A checking account component may have the responsibility of "knowing the account balance". Alternatively, it may have the responsibility of "knowing how to get the account balance". The second may imply to some people that it does not store a current copy, but calculates it, while the first may imply to some people that it stores a current copy. In fact, there is no difference, since it might "know the balance" by computing it, or might "know how to get it" by storing it locally. The difference corresponds to the difference between direct properties and derived properties.
- Easing the difference between "knowing" and "knowing how to" is deliberate. The decision of which way to implement the responsibility is a design decision to be addressed at a separate time and with other design concerns in mind.
- When the design team gets skilled with responsibilities, they may work by writing down only the most important or summary responsibilities, and fill in the rest when they finally document the component. However, in getting started, a team may want to write down everything, not to forget. It is useful then to keep the active responsibilities at the top, since they are key to partitioning the system.

Partitioning responsibilities

- ***A responsibility often consists of other responsibilities***, since a service consists of other services.
- From the point of view of applying the technique, either the summary or the detailed list of responsibilities may be used in partitioning. At the beginning, the design team may wish to work with the detailed list of services to be sure no gaps are present. Eventually, they should work with the summary statement, as it is much faster. The detailed list of services will be worked out eventually.
- For example, a bank account may have the responsibility of handling and tracking all the transactions to that account. The individual services are add a transaction, remove a transaction, create a transaction to handle monthly charges or interest calculations, etc. Writing "handle and track transactions to the account" is easier to write, read and work with while the responsibilities are being partitioned. The complete list can be created over time.

B.4 DISCUSSION OF COMPONENTS

Two categories of components

- The creation of a component is usually an assertion of one of two things:
  1. The component represents ***something tracked and managed by the business***,
  2. The component is a point of ***design variation***, capturing the common characteristics of several possible solutions.
- (1) If the business manages "customers" and "orders", then Customer and Order need to show up as components at some stage of the design. Their absence would mean that the design is not complete, that they will show up at another level of discussion, or that the term is just a nickname for some other thing that the business really manages.

- (2) Often, components are created as placeholders for one from a choice of possibilities. So often, that the design will be checked to see that there is a possible alternative implementation for each component. One of the strengths of responsibility-based modeling and object-oriented implementation is that a component represents a set of services behind which various implementations may hide. Defining those services as a component allows the designers to vary the implementation over time, without change to the users of the services. That is, the component serves as a *point of design variation*.
- The value of using components as points of design variation cannot be emphasized enough. A component is checked to see whether it represents a class of implementations that are likely to appear over time. In addition, the designers are advised to consider creating a new component whenever there are possible variations in the requirements or the implementation. The new component will characterize the services required, and make where the decisions can be varied, providing future safety and permitting the design session to continue.

A final word on "component"

- This technique is largely targeted toward object-oriented designs, in which the final design is implemented with classes and instances of those classes. Such an instance is a component, and the class defines those components. However, not all of the components in the system are instances of classes. Some of the components at the lowest level may be programming, operating system or network or database services. At the end of this technique, they must be specified. Above the lowest level, a component may be a collection of object types and instances that have to work together. For the purposes here, that collection may be treated as a single "thing", a component that must be decomposed further. A framework of any sort may be treated as a component; a subsystem may be a component in a larger system. If the project is not an application development project, none of the components may end up being OO classes at all, but collections of people and programs. For these reasons, the word "component" is used throughout, until the very end, when type specifications must be produced for those components that are types and type instances.

B.5 CREATING VS. REUSING COMPONENTS

- This technique works toward the use of existing components. The technique terminates whenever all the scenarios can be delivered using a combination of existing and newly specified components. This is called "design with reuse" (as contrasted with "design for reuse", which hopes some other project will use the results of this design). Not much of great use can be said about finding the best components to be used or reused except for this:
- ***It is the responsibility of the designers using this technique to identify the best set of existing components to use***.
- Real productivity gains come from using existing components. It is the responsibility of the design team at this point to be aware of the kinds of components that can be pressed into service. Sometimes a component can be found that nearly does what is needed. It becomes a design issue whether
  - to ***use that component directly***,
  - ***create a new component that uses*** it (remember composition technique) or
  - ***not use*** it.
- When a needed component name is described in terms of its role, look for an existing component that has those capabilities already. It may turn out that the existing component can be used, either directly or indirectly.

- The ***trade-off*** between reusing an existing component for its capability and introducing a new, specific component is the trade-off that must be managed by the partitioning team. The choice is made by sensing the cost of changing client code versus the cost of introducing a new component to the system. It helps to make that cost comparison explicit. The new component must be designed, tested, documented, maintained, and learned by future designers

One example, two outcomes:

- The account needs a journal having responsibilities to add and remove transactions, and mediate transaction history. The team names a card, "*Journal*", and then decides that an existing object type, "*OrderedCollection*" could be used directly.

  Outcome 1: ***Reuse****.*

  - This application is not for a bank. The account is a relatively minor part of the system, and only a few places use the account and its journal. All services required of the journal are made the responsibility of the enclosing component, the account. The account will print the journal, find entries before or after a certain date, make a copy of the journal, etc No new component is created for Journal. The card for Journal is annotated with "*OrderedCollection*" to indicate it already exists there.

  Outcome 2: ***Create*** *(with some reuse).*

  - The team decides that the concept of a journal is important to the business, that it will be heavily used and that the representation of the journal might need to change over time (for performance or because new responsibilities might be added). They create a component called *Journal*, enumerating its services: present itself in various ways, find entries before or after a certain date, etc. The object type designers eventually look at the requirements and decide it will use an *OrderedCollection* to hold the data (initially), but that is a later design issue.

General Hints:

- If the component is a major concept in the model, widely used, and is likely to change, then ***create a new component***, and let it be implemented by one of the available components. The cost of introducing a new component is likely to be less than the cost of changing the client code. Mark the existing component as a collaborator.
- If the new component needs only a subset of the capabilities of an existing one, it may be better off as a separate component. The additional services offered by the existing component may be a hazard to the component needed. The new component can conceal the inappropriate part of the existing component's interface. Mark the existing component as a collaborator.
- If the component is not a major concept of the model, and is either not used widely or not very likely to change, ***reuse the capability-named component directly***. The cost of change is likely to be small compared to the cost of introducing the new component to the system. Consider moving some of the responsibilities to the enclosing component.

B.6 DISCUSSION OF LEVELS AND SUBSYSTEMS

- Since a component will often consist of other components, it is important to control which ones are in the discussion, and which are outside. Consider a set of components that read and interpret text typed by the user. At one level of discussion, that set of components is a single thing, a subsystem of lower-level components. At this level of discussion, it is sufficient to treat the set as one component, and discuss the responsibilities of that component in the system at large. If there is some question as to whether such a subsystem

could actually be designed, or its performance, that component may be unfolded into its sub-components, and examined. After that examination, it is important to hide or fold back together the sub-components and work with the subsystem again as a single component. Managing the number of components under active discussion is key to working with the technique.

- A subsystem is any collection of components with a unified purpose. Different subsystems can cut across the system in different ways, producing overlapping component groupings. The "user interface" components, for example, form a subsystem. The "network" component does likewise. Depending on the level of discussion, the "customer" may be treated as a single component, or dissected as a set of components. A consequence of working with different levels and with subsystems is that two components sometimes appear to have the same responsibility. The two components operate at different levels, an outer and an inner. At the outer level, the inner component is not visible, so it cannot be sent a message. It is not "visible" to the other components at the outer level. So a component at the outer level acts as a sort of gatekeeper, or contact point for that responsibility. It then just delegates the responsibility to the other, inner component. This delegation of responsibility is appropriate. The outer component is acting as a subsystem, appearing as a single component at the outer level, and as a member of a set of components at the inner level.
- Consider a bank account. It has the responsibility to track its transactions. On closer examination, the account turns out to have a collaborator, a "journal" component, whose responsibility is doing the actual tracking of the transactions. The discussion of the journal as a separate component may not be appropriate at the level of discussion in which the account is a single component. In fact, the decision as to whether the journal is a separate entity at all may be a design decision that changes over time. To protect that decision, the account is given the responsibility to track transactions. Whether it tracks them itself or delegates that to the journal is its own design decision, not visible at the outer level.

B.7 DISCUSSION OF INHERITANCE AND POLYMORPHISM

Inheritance

- The decision whether to use inheritance is only partly made in responsibility-based modeling. It is nominated in this technique, and finalized later, when the component is finalized, or in framework design. The recommendation to consider inheritance can be made from responsibility-based modeling based upon common services required across similar components.
- A set of responsibilities shared over a variety of components may be collected into a new component, a generic version. The new collection of responsibilities may turn out to be a separate kind of component in its own right, and not a generic version of the components that contributed the responsibility. The new component must be evaluated for its stability and contribution just as any other.
- If the generic component survives, it may be cast into implementation in one of several ways.
  - The specific components may send it messages asking for the common service (delegation),
  - The specific components may inherit its services (inheritance).
- ***Inhertiance*** is a heavyweight relationship between two components. It is not always the best choice for implementing the relationship between generic and specific components. Nor do all implementation technologies support inheritance. Therefore, the decision to use inheritance, delegation or some other implementation technique is left as a choice for the

component designers. The team may prepare suggestions on the similarities between components that will help the designer.

Polymorphism

- Polymorphism is the OO term used to express the fact that two components provide similar services, e.g., an order line item has a value and so does the order itself. An order may be asked to provide its value, which it may do by asking each line item for its value, and then adding them together and altering the sum according to tax laws, etc. The service, "provide its value", is polymorphic between order and order line item. "Value", as a verb, is particularly varied, since many different things can have and describe their value in many ways.
- Polymorphism provides a savings in conceptual complexity. The fact that the same verb phrase is used for several components to carry the same intention, even if the implementation of each is different, means that fewer verb concepts have to be learned to understand the design of the system and its implementation. The partitioning team should consider the value of polymorphism when naming responsibilities.

B.8 SCENARIOS AND RESPONSIBILITIES

Relationship between scenarios and responsibilities

- Scenarios and responsibility allocation are closely related.
- A ***scenario*** is characterized by its goal, which the primary actor *wishes to accomplish* with the system. The system, on its side, promises to carry out certain functions, which, if it does, allows the actor to accomplish the intent. For example, a bank employee wishes to "register a customer's transaction". That is the actor's goal. The requirements team also gives the system the responsibility to log the transaction by its date, and update and log the account balance. Those responsibilities show up in the scenario statement.
- In RBM, the ***system is partitioned into components that carry out the system's responsibilities***. The interactions between the components are documented. To each component, the ***interaction between it and its collaborators appears as a scenario***!
- That is, *an actor requests a service or initiates some sequence of related messages that the component must respond to. When that component undergoes design, each of those requests and message sequences will be treated as scenarios for the component*.
- This process of "scenario - responsibility – interactions" repeats itself at increasingly specific and detailed levels until one of these things happen:
  - A component is found that can carry out the responsibilities. This component may be a object type, an external service (such as an operating system, database or network service), a complex subsystem, or even a human organization or person.
  - The level of "object type" (in the object-oriented sense) is reached.
  - The level is reached of a non-objected-oriented service that must be designed and implemented. This service is designed using a suitable design technique for the implementation technology, such as organizational design for components consisting of people.
- Here, the ***level*** is the nature of the concerns at a certain point in discussing a design. A component is visible at a ***level*** if it has responsibilities relevant to the nature of the discussion at the moment.
- Thus, scenarios and responsibilities allocation go together to make a complete manner of design, and the degrees of freedom resolved by the design technique are the names of the

components, their responsibilities and the way they work together to deliver the required system function.

Interaction diagrams

- Functional equivalents of an interaction diagram may be written in text, drawn as a list of horizontal arrows, or drawn with a graphics editor or specialized tool. If drawn, it may be drawn in topological view or time view.
- An interaction diagram describes the sequence of interactions between components in resolving a particular situation. In the textual form, the sentences are listed in order of occurrence, one interaction per sentence. In the time view, each interaction is represented as an arrow going from the message sender's column to the message receiver's column. The interactions are listed in order of occurrence. In the time view, parallel or unordered activities can be shown. In the topological ("top") view, the components are laid out on the page however the author wishes. An interaction is shown as an arrow gong from sender to receiver. Each arrow must be numbered to show the sequence.

B.9 RDD / RBM STEPS

Overview:

1. Identify scenarios to use; bound the scope of design. Identify the scenarios in the scope. Order the scenarios to apply. Work with the main scenario first, using the alternate scenarios as variations.
2. Role play the scenarios, evaluating responsibilities.
3. Name at each point the responsibility needed to carry the scenario toward conclusion. Name an existing component or create a new component to carry the responsibility. Point to the business model, object instance diagram CRC card, or whatever is holding the design discussion.
4. Make sure that each service provider has sufficient information and ability to carry out its responsibilities.
5. Consider variations of the scenario to check for the stability of the responsibility allocation. Play through the original scenario again to verify it works.
6. Evaluate the components with test questions and variation analysis.
7. Ask whether each component protects against future changes or is something the business manages directly. Check the life cycle of the components: creation and deletion.
8. Create variations: ways the requirements or implementation might change over the life of the system; alternate path scenarios and error conditions.
9. Run through the variant scenarios to investigate the stability of the components and responsibilities. Revise as needed to strengthen the design.
10. Simulate if possible.
11. Consolidate the components by level.
12. If a set of components are at different levels of abstraction, note to which primary card they are related and their purpose. Design that set of cards later as a subsystem. Give a mnemonic name to the scenarios requiring those cards, for easy recall at that later time.
13. Document the design rationale and handling of key scenarios. Document either at the end or just after the handling of a set of related scenarios has become stable.
14. Decide which scenarios to document (main, error, interesting ones)
15. Document each selected scenario and why responsibilities were allocated that way.

16. List the components being used that already exist.
17. Specify each new component of the design session's level.

Collect relevant scenarios.

- Bound the scope of design in width and depth (level). Settle in advance what is being designed, what is not, and how to know when the design is done. Settle at what level the design is being carried out, and what topics are relevant to the design.
- Gather together and read the scenarios relating to the current scope of design. At the outermost system level, choose scenarios developed with users. If an internal subsystem, work from the scenarios, responsibilities, and interaction diagrams created in the previous design sessions. If there are no scenarios and this is entry to the project, go through scenario design.
- Choose first a simple scenario that sets up parts of the system. Increase to more difficult ones. The more difficult ones expose more decision points and should be reached as soon as possible.
- If you are comfortable with both the problem domain and CRC cards, choose a scenario handling a more complex situation first. Choose next a series of scenarios in the order they apply to the subsystem.

Choose the most complex scenario of each similar group.

- The intention is to reveal the most decision points, and to show up the most complex collaborations between components. More complex situations do this faster.
- If the most complex situation proves too difficult, back up to a less difficult situation, but get back to the difficult case again as soon as possible.
- Select the main success scenario to use first. The main scenario is the one that delivers the primary actor's goal in the most direct fashion. Use the alternate scenarios as variations within a single walkthrough and role play, as appropriate. Any that are not covered in the role play, apply to a separate walkthrough and role play. Treat an alternative scenario as a variation during walkthrough of the main scenario if it reveals interesting decisions in the components being used in the main scenario. Treat an alternative scenario separately if requires invention of components that do not show up in the main scenario. Apply the failure scenarios as tests to the design for the related success scenarios.

Identify components and their responsibilities for each scenario.

1. *Use a combination of active and contact point responsibilities. Have active responsibilities start with an active verb.*

- Examples:
  "compute new balance",
  "find all customers with given characteristics",
  "refresh the screen",
  "maintain consistency of customer addresses."
  "introduce transaction."

2. *State active responsibilities in a generic form, so similar responsibilities can be identified.*

- Example:
  Better: "display", or "display on screen".
  Worse: "display triangle on screen."
- Reasoning. When the design is done, it is likely that a number of components will say, "Display on screen." In the review of the responsibilities, these will be readily spotted. A review of these similar responsibilities will result in a design decision between:

- there is a generic component that has not been identified, and to whom these can all delegate their responsibility, Mark it as a possible place for inheritance.
- there is no generic component. Rather, there is polymorphic behavior. The responsibilities should be reviewed again to make sure that they carry the same intent, and that a client is able to make a safe assumption about the behavior of the component when calling upon that responsibility.

3. *Identify the information the component mediates (for which it is the contact point).*

- Discuss contact points instead "state data" or "attributes". It may not yet be time to decide that a component must own certain data or attributes. The implementation is likely to evolve over time, invalidating those assumptions. It is possible that the properties get separated out from the original component, i.e., the component will get unrolled.
- **Example:** Knowing the balance on an account is a contact point responsibility of the account. It may come to pass that the balance is computed dynamically. Recording the bank account is a contact point for the balance
  - (a) records the necessary responsibility,
  - (b) leaves the decision open as to how and when the balance is computed and where and whether the result is stored.
- Be alert for the opportunity to reuse an existing component. The design team is responsible for considering the various options of reuse versus new component creation. Whenever a component name describes its *role* instead of its *capabilities*, look for an existing component that already has those capabilities. Decide whether to create a new component, protecting the design, or reuse the existing component, saving development, based on the number of other components that reference it and the likelihood of change.

4. *General hints*

- If the component is a major concept in the model, widely used, and is likely to change, then create a new component, and let it be implemented by one of the available components. The cost of changing components that reference it is likely to be greater than that of introducing a new component. Mark the existing component as a collaborator if appropriate.
- If the new component needs only a subset of the capabilities of the existing one, it may be better to introduce the new component. The additional services offered by the existing component may be a hazard to the component needed. The new component can conceal the inappropriate part of the existing component's interface. Mark the existing component as a collaborator.
- If the component is not a major concept of the model, and is either not used widely or not very likely to change, reuse the capability-named component directly. The cost of change is likely to be small compared to the cost of introducing the new component to the system. Consider moving some of the responsibilities to the enclosing component.

5. *Component name*

- It often happens that there are competing nominations for a component, or component's name. Sometimes the team agrees that one of the two names carries the needed responsibilities better than the other, and so the second nominee can be dropped. On occasion, neither nomination completely dominates the other, but rather, the two alternate in usefulness. In this latter case, the design team needs to explore whether there is a common abstraction lurking in the background, or perhaps a third abstraction that can pull out common elements of the two, or whether there is a miscommunication about the two names.

6. *Components protecting design variations*.

- Discussion may arise whether some part of the system will evolve "this way" or "that way". Often there is a meaningful abstraction that carries responsibilities and services that apply to all of the suggested implementation alternatives. That abstraction becomes a key component that ***protects a design decision***. It protects the right to change the decision later or even dynamically at run time. It defines a necessary service interface that the clients of the component need, regardless of the implementation. The identification of such design points is critical to ensuring the stability and robustness of the system over time, and is key factor to responsibility-driven, object-based, and object-oriented systems.
- Example: In the design of a map system, the team got embroiled in a discussion of whether the routes are going to be pre-computed, dynamically computed, how they will be represented, etc. At this point, the component "Routing Strategy" was introduced. The Routing Strategy component has the responsibility to obtain a route through a list of locations. Once this component is created, the topic of computing and storing routes can be deferred. In fact, in an initial prototype, a pre-selected list of routes can be hard-coded in the Routing Strategy component. In initial versions, the routes can be computed on the fly. In later versions, there can be a mix of dynamically computed routes and precomputed routes sitting on a database. The Routing Strategy object defined the services that are common to all implementations, and permits the growth of the system over time.

7. *Design Patterns*.

- Design patterns (such as ***strategy***) contain many ideas for protecting design decisions.

8. *Homogeneous collections.*

- Collections of objects of the same type show up repeatedly, at all levels of design. Sometimes the items in the collection are relatively uninteresting, and the interesting behavior shows up in the collection (e.g., ledger lines are dull, collections of ledger lines are interesting). Sometimes the collections are oriented toward the user interface (e.g., list boxes), sometimes they are oriented toward the database (e.g., persistent collections). Be prepared to introduce homogeneous collections your components. Many of these collections are already named in the essential business relations.
- When working with a collection, pay attention to which component "owns" it, creates it, initializes it, keeps knowledge of its contents consistent (e.g., supposing there are many other components in various stages of browsing or updating it).
- Sometimes good abstractions are discovered after rather than before design. That is, only after working with the scenarios and components for a while does it become apparent that several components could be viewed as variants of a generic abstraction, or that they could fruitfully delegate to another abstraction.
- If the components having common responsibilities otherwise have little in common, consider introducing a new component to which these components delegate their common responsibilities.
- If the components seem to be variants on a common theme, or the responsibility just cannot be delegated further, look for the abstraction that could be the generic component.

Be aware of some common situations.

9. *Delegating to a different level.*

- A component may delegate its responsibility to a component at a different level. It will appear that two cards have the same responsibility. The difference between the two components is that the second one is not properly known to the component sending the request to the first.

10. *Difficulty in allocating a responsibility (too many places or no places).*

- Responsibility appears to belong in too many places:
- Some component has been too broadly specified. Reconsider each component, the abstraction it represents, its purpose in the system. Try to decide whether one of the components has been too broadly specified. It is possible that in its new definition, it no longer answers 'yes' to the question, Is it really this component's responsibility to handle the responsibility?

11. *A new component needs to be named.*

- The reason that the responsibility appears to have many homes is that there is an unnamed component lurking in the background that needs to be named. Once it is brought out, it can take the common part of the responsibility.
- **Example**: Train engines are hooked to cars and a caboose. There is difficulty in deciding whether hooking the engine to the caboose the responsibility of the engine or of the caboose.
- **Solution:** There are missing abstractions, that of a train configuration, and that of a train. The configuration of engine, cars and caboose is likely to be something that the system will want to manage, and want to vary over time. It may be correct to introduce "train configuration" (or the equivalent, correct term from the transportation industry). It is correct to give the train configuration the responsibility to register the addition of each engine, car and caboose.
- A design decision must simply be made. Sometimes there is simply a choice that must be made. There are cases where there are no further things to be considered, and two components appear equally well suited to the task. The design will probably survive either decision. Later developments may reveal which was to be preferred.
- **Example**: A warehouse contains boxes. Does the warehouse know the location of every box, or does each box know its own location? Each way can be made to work, each has its characteristics, neither is clearly "better".

12. *Responsibility appears to belong nowhere:*

- Some component is too narrowly specified. Reconsider each component, the abstraction it represents, and its purpose in the system. Try to decide whether one of the components has been too narrowly specified. It is possible that in its new definition, it answers 'yes' to the question, Is it really this component's responsibility to handle the responsibility?
- A new component needs to be named. The reason that the responsibility appears to belong nowhere is that there is an unnamed component that needs to be found. Once it is found, the abstraction is represents can be seen and named.

Evaluate components and responsibilities.

· Consider alternative assumptions and scenarios to check and improve the design. Hypothesize additional scenarios as needed.

1. *Check alternate scenarios for variations in outcome, creation and error conditions.*

- For each scenario, walk through every variation to check that the responsibilities named and their allocations work correctly. For the first several, it is likely that new components will have to be named. After a while, the scenarios will use the named components in a very obvious way, so that the walkthrough will be brief.

2. *Completeness criteria.*

- It is not necessary to walk through every variation of every data condition. It is only necessary to walk through scenarios that reveal something new about the working of the

components. Once the walkthrough of a scenario reaches a point that has already been discussed in depth, the facilitator may say, "... and we have established that that works". If that point establishes the correct delivery of the scenario, the scenario is ended. If the scenario continues on later in a different way, the walkthrough picks up at the point at which something new happens.

- Consideration of initial scenarios and error causing scenarios is important. The error condition scenarios define against what errors the system must be able to protect itself. Initialization scenarios reveal which components are able to create which others. Both need to be documented.
- **Examples**: In car rental: the client rents a car, crosses a time zone and returns the car before it was rented. In banking: a new customer requests an ATM card.

3. *Check likely requirements variations.*

- Consider the evolution of the system. What sorts of enhancements might be requested by the customer after using the system. Typically, the initial requirements statement contains simplifying assumptions for the first version, which will be changed several versions later. The walkthroughs are well suited to discussing some of the likely future enhancements or requirements changes. A new component can be put into place to protect against a particular requirement change (as per "Components protecting design variations", above). The component will allow the first system to be built with the simplifying assumption, but contain a place for the enhancement.
- The design team will have to decide what requirements variations to consider within the scope of the design effort. A requirements variation such as, "consider including everyone in the world" may not make sense for a local office system, but may for a telephone system.
- **Example**: In the map system example used earlier, the requirements may say, "obtain a route between two locations from the database". The team is concerned
  - (a) that a future enhancement will be to have a route go through multiple locations (e.g., "find a route from Boston to L.A. through Chicago and Santa Fe),
  - (b) that a future enhancement will be to use routes computed dynamically.
- The team introduces a component, "List of locations", instead of always assuming two locations, and a component, "Routing Strategy", that conceals whether routes are looked up or computed. Initially, the routing algorithms will only work for two locations in the list, and will always look them up in the database. When the later enhancements are requested, the use of the list can be expanded, and alternative classes implementing different routing strategies may be added.

4. *Likely implementation variations.*

- Likely variations include looking up answers versus computing them, and likely changes in the business.
- Validate each component's name and responsibilities.
- "***Is it really this component's responsibility to handle these requests?***"
  - Your intuition is your guide to what responsibilities go with the abstraction the component represents. Use your intuition and your judgment.
- "***Does this component already exist?***"
  - Look into the components and design pattern catalog again to see if an appropriate component can be found, knowing what you know now.
- "***Does this component have too few responsibilities?***"

- Be alert for a component that is just a glorified responsibility. A component is supposed to capture an abstraction that has a purpose in the system. It may happen that what appears at one moment as a meaningful component is really just a single responsibility left on its own. That responsibility could be assigned to a component.
- Alternatively, the few responsibilities characterize a "role" that a component can play. Look for a component that can play that role.
- Look for another component that does similar work, see which of the two components carries the better abstraction for the system, and see whether one of the two components can be eliminated.

- "***Does this component have too many responsibilities?***"
  - Be alert for a component that ends up as a kitchen sink full of responsibilities. A component is supposed to capture an abstraction that has a purpose in the system. A component with too many responsibilities may not be a single abstraction, but several, mixed together. A complex component is harder to reuse than a simpler one that captures an abstraction in a purer way.
  - A place to allow lots of responsibilities is a large subsystem in a large system. A preferable reason for creating a subsystem may be that it has a single or a few key responsibilities. However, there may be other valid reasons to create a subsystem, which leave it with a bag of responsibilities.
- "***Does this component protect design decisions? future subtypeing? implementation variations?***"
  - Many components either deliberately protect design decisions as described earlier, or can support multiple variants or implementations.
  - If a component does not do one of these: Perhaps it is a necessary item the business has to manage and hence necessary to keep. Perhaps its responsibilities might fit somewhere else, allowing it to be removed. Perhaps it might be better named, so that a better abstraction will surface.
- "***Does the name accurately reflect the abstraction and the capabilities?***" "***How easy will it be to find and use the component by its name when it is viewed out of the context of this system, during later reuse?***"
  - Names are terribly important. They are what people focus on when understanding the system and looking for existing components to use.
  - "You have to be able to make assumptions about a component based on its name." (Ward Cunningham).
  - "If your teammate can't make assumptions about your code, you are just laying grass over quicksand." (Hayden Lindsay).
- *Names and naming habits to use*:
  - Name according to the abstraction represented, **e.g.,** bankingTransaction. Name according to its capabilities, not its role in the system.
- *Names and naming habits to avoid*:
  - "manager". Consider "broker", "librarian", or similar.
  - "-er" suffix when applied to a formula. Consider "policy" or "strategy". Example: Responsibility is to obtain a formula for a rate. Try "Rating Policy" or "Rating Strategy" instead of "Rater".
  - "data". Try to find the abstraction it represents.
- "***What other component(s) control the life cycle of the component?***"

- How, when, by whom is it created?
- How, when, by whom is it destroyed or deleted?
- Make sure these questions have answers available in the partitioning, otherwise go back and handle them.

- "***What changes in available behavior does it go through and how are those handled?***"
  - **Example**: A credit card corporation: A customer goes from being a prospect to a plain cardholder to a preferred cardholder, possibly to a delinquent cardholder, to a former cardholder. Each of these "states" of being a customer of the credit card corporation has different behavior associated with it. The design must account for migration between states and the different capabilities that come with it.
  - Entities that change behavior over time are common in business systems. Be alert for them. The changes in behavior that come with changes in states are not handled easily by current technology. They must be designed deliberately and carefully.
- "***Are the responsibilities phrased in active terms?***"
  - Question any component that simply acts as a contact point for information. Some of these are needed (**e.g.**, a Banking Transaction may only be used to hold the transaction data together), but sometimes they can be given greater responsibilities.
  - "It takes fewer magic carpets to cover an application than it does rugs." (Rebecca Wirfs-Brock). A component with active responsibilities is like a magic carpet. A component with no active responsibilities is like a rug.
  - use words like:
    "Obtain..."
    "Add...", "Remove...", "Introduce..."
    "Sort..."
  - Try to avoid words coming from the computer profession terms and words like:
    "Hold..."
    "Manage..." (this word is occasionally necessary)
    "Know..." (there is a separate place for "contact point" information)
- "***Is this component implementable?***"
  - Do check down a level to make sure a component is implementable. The design team declares, with the design, "If we had these components, we could deliver the needed function in this fashion." With that declaration there comes an assertion that the components are available or can be built.
- Check the pattern of communications.
  - Are the communications intensive around one component?
  - If one component dominates the communication pattern, it is possible that it has too many responsibilities and knows too much about too many parts of the system (it is a "manager"). This could become a fragile part of the system: if any of the parts it knows about changes, it may have to change. If it is a large and complex component, then the chances of introducing an error or propagating changes is higher. Consider distributing its responsibilities.
- *"**Are the communications intensive between two components?**"*
  - Two components communicating with each other intensively evidently need to know a lot about each other.
  - Perhaps they ought to be combined into one component. **Example:** In Smalltalk/V, the *Model-View-Controller* separation had up with intense

communications between the *View* and the *Controller* for GUIs. By popular demand, the *View* and *Controller* were combined to make an *Interactor*. The new abstraction was considered by many easier to use.

- Perhaps there is a third abstraction waiting to be found. Sometimes the communications represent the information and responsibilities of another component, which has not been named and is split across the two components.

· At this point you should have an improved stack of cards, labeled with responsibilities, that carry out the required design better or requiring less new design.

· At the end of the design session, when the most complex scenario has been designed to satisfaction, prototype the design to check the flow and improve the evaluation of the design.

· Often, implementing the scenario will reveal that the hand-off of responsibilities is not perfect, or that information is not available at a point where it is needed. Finding such a situation during the design period pays off. Consider the computer another member of the design team, able to offer feedback on the design.

· Implement just the components needed, running from the most available user interface (e.g., direct program control, Transcript Window, or equivalent).

· Walk through the execution of the system to check that all the information and responsibilities flow correctly.

· Use the implementation to validate or even create the documentation for the scenario. The implementation is unambiguous and guaranteed to show all the information that has to pass from one component to another. Some tools support the creation of interaction diagrams from the execution trace.

Document the design rationale and key scenarios.

· Keep the main scenario of each scenario. Keep the error scenarios of each scenario. Keep any other scenario that has an interesting rationale, or shows non-obvious communication between components. This is a matter of judgment on the part of the design team and the documentation requirements of the project (traceability). Too much documentation is burdensome without commensurate value. Too little documentation causes confusion.

· Document each selected scenario. Create an interaction or object instance diagram using the components at the declared level. Give each scenario and interaction diagram a name. If several scenarios have a set of interactions in common, consider making a sub-scenario with interaction diagram(s) to describe the common part. Then reference the sub-scenario in the interaction diagrams having those interactions in common. The sub-scenario acts is a scenario at a level too detailed to have been mentioned earlier. The use of sub-scenarios shortens the documentation considerably without loss of detail.

· The interaction diagram must show the scenario from its beginning to the completion of the scenario. The purpose of the diagram is to declare how the components cooperate to deliver the required function (even if the function is only returning an error condition). Therefore, the diagram must start with the initial message that starts the scenario, and end with the final resolution of the scenario.

· Describe why the responsibilities are partitioned they way they are, if there is a business or non-obvious reason. The partitioning of the responsibilities may reflect some basic business process or assumption. This information is easy to lose and useful to come back to later. Write it down at the top of the interaction diagram and eventually as a comment in the program code.

- **Example**: "The computation of the insurance of three houses in different zones is based upon primary house. The rate for the insurance is based on the rate for the primary

house modified by the rates for the secondary houses. Responsibility for computing a base rate is given to the primary house. Responsibility for modifying a rate is given to a secondary house, which returns the new rate for the previous houses plus itself."

- Sometimes a lot of work went into allocating the responsibilities. The thinking behind the allocation should not be lost, as it is likely not to be obvious to other people.
- Example: "*The routing strategy component exists to preserve the freedom to choose between stored and computed routes. It has the responsibility to obtain a route; however that route might be stored or computed."*
- Sometimes the names of the components make the allocation of responsibilities obvious. These need not be documented.

B.10 CRC CARD EXERCISE

Overview

- A CRC cards is an index card that is use to represent the responsibilities of classes and the interaction between the classes. CRC cards are an informal approach to object oriented modeling. The cards are created through scenarios, based on the system requirements that model the behavior of the system. The name CRC comes from Class, Responsibilities, and Collaborators which the creators found to be the essential dimensions of object oriented modeling.
- CRC cards where introduced by Kent Beck and Ward Cunningham in there paper "A Laboratory for Teaching Object-Oriented Thinking" released in OOPLSA '89. There original purpose was to teach programmers the object-oriented paradigm. When Kent Beck wrote the draft version of their paper he changed Collaborators to helpers. Ward Cunningham changed it back to Collaborators when he reviewed the paper. The initials of Cunningham's son are CRC.
- In a CRC exercise, a card is made to represent an instance of an object type. Its responsibility is identified, either by invention or writing it from the object type definition. A use case scenario is begun. Someone talks through the scenario, and one or more people show the objects that work together to deliver the scenario. When one object uses another, the second object is said to be the first object's collaborator. The names, the responsibilities, and the collaborations summarize the design at a low but accurate level of precision.

The Group

- The ideal group size for a CRC card session is five or six people. This size generally allows everyone to productively participate. In groups of large size the productive is cut by more disagreements and the amount of participation by everyone is lower. If there are more than six people, one solution is to have the extra people be present strictly as observers.

The Card

- The cards should look something like shown in the Figure 10.
- The exact format of the card can be customized to the preferences of the group, but the minimal required information is the name of the class, its responsibilities and the collaborators. The back of the card can be used for a description of the class. During the design phase attributes of the class can be recorded on the back as well. One way to think of the card is the front as the public information, and the back as the encapsulated, implementation details.

Component name	
Synopsis	
Responsibilities	**Collaborators**
active responsibilities	*component names*
...	...
...	...
...	...
contact point responsibilities	...
...	...
...	...

Figure 10. CRC card outlook.

- The component name is written across the top. The responsibilities are written in three groups.
- First is a brief summary, or synopsis, of the responsibilities of the component, its role in the system. This should be a short phrase, or perhaps two.
- Next, in a list down the left side of the card are the active responsibilities, with a line or arrow to the right, ending in the name of a required collaborator for that responsibility. An active responsibility starts with an active verb, such as "track", "compute" or "find". Avoid the word "manage" where possible, and the passive verb, "hold".
- Last are the contact point responsibilities, the information the component mediates. Often these will come from the attributes in a data or business model. If there is some question whether a service belongs in the active or contact point responsibility section, choose arbitrarily with a slight inclination toward the contact point section. It really does not matter a great deal. In the end, all responsibilities will be treated equally. The purpose in having the sections is so that attention can be focused on the summary and active responsibilities, which are the primary vehicle for partitioning the system. The contact point responsibilities are needed for component specification, and to demonstrate how the components deliver the required function in a documented scenario.

The session

- Use a centrally visible and accessible table.
- Have available a stack of blank CRC cards (see Figure, "CRC Card"). Place them on the table within reach of anyone or give everyone a set for themselves.
- Before starting a session there needs to be some kinds of requirements for the systems. Weather they are implicitly or explicitly define the people participating in the group need to be familiar with them. A session also needs to focus on one part of the problem at a time. So a subset of the problem needs to be chosen to explore during the CRC card session.

1. *Creating class*

   - The first step in modeling a system in the object-oriented paradigm is to identify the class in the problem domain. So this is the first step in a CRC card session. Using the problem statement or requirements document, identify the classes that are obvious in the subset of the problem that is going to be explored in this session. One useful tool is to find all of the nouns and verbs in the problem statement. The nouns are a good key to what class are in the system, and the verbs show what there responsibilities are going to be.

- Use this information for the basis of a brainstorming session and identify all the class that you see. Remember in a brainstorming session there should be no or little discussion of the ideas. Record them and filter the results after the brainstorming. After the classes have been chosen pass out cards and assign the class to the member of the group. Each person should be responsible for at least on class. They are the owner of that class for the session. Each person records the name of their class on a card. One class per card.

2. *Responsibilities*

- Once a reasonable set of classes have be assigned to the group, responsibilities can be added. Add responsibilities that are obvious from the requirements or the name of the class. You don't need to find them all or any. The scenarios will make them more obvious. The advantage of finding some in the beginning is that it help provide a starting place.

3. *Scenario execution*

- These are the heart of the CRC card session. Scenarios are walkthroughs of the functions of the system in detail. Take required functionality from the requirements document and use this as a scenario. Start with scenarios that are part of the systems normal operation first, and then exceptional scenarios, like error recover, later.
- First decide which class is responsible for this function. The owner of the class then picks up his card. When a card is in the air it is an object and can do things. The own announces that he needs to fulfill his responsibility. The responsibility is refined in to smaller tasks if possible. These smaller tasks can be fulfilled be the object is appropriate or they can be fulfilled be interacting with other objects. If no other appropriate class exists, maybe you need to make one. This is the fundamental procedure of the scenario execution.
- Pretend the design participants are the individual components, and have to deliver the function in the way the scenario says. The people role-play the components. On feeling embarrassed. It feels odd to begin with, pretending to be a component and not knowing what the component should do. Although it may feel odd, it is by pretending to be the component that a person can best address whether a responsibility is correct or not.
- Ask what kind of component should handle the entry. Pick up the card for that component. Hypothesize the responsibility and the component, inventing new ones if necessary.
  - *Note:* Here is your big opportunity for reuse. Use things that exist, if possible.
- Identify what the component would need to get its job done. Look into the catalog of existing components for a component that already does it. If none does, carry on asking what kind of component should have the needed responsibility.
- Continue in this way until the scenario reaches its conclusion, using the responsibility-based modeling technique.
- **Tip**: When things get moving rapidly, sometimes there is no time to write down the name of the responsibility or the name of the component. At those times, just point to an existing card or even to a blank spot on the table, either naming the component or just saying, "This one". If the design works, the component will show up consistently, and a good name can be discovered for it and its responsibilities. A good name is so important that it is worth delaying the naming of a component until its purpose is clear and agreed upon.
- **Tip**: Pay attention to the level of the discussion. As mentioned above, it is fine to go deeper than the level of the design session periodically. The components nominated at other levels may become elements of subsystems that will eventually be designed. Note the components that belong to a different level, and either stack them under the card that

is calling for their use, or set them to the side, so that they can be pointed to or reintroduced when there is a question.

- **Tip**: Only use one pen. That way, it does not happen that one person changes the name, responsibilities or collaborators without the rest of the team noticing. The pen acts as a synchronization mechanism for the group.
- The role play is an act of creation, in which design points are discovered, components and responsibilities are nominated, and design decisions are made. A choice between two names, between the need for a component or not, between two places to allocate a responsibility, is made is made by comparing the two choices against a set of scenarios. That choice is preferred which responds best to the scenarios. The choice is made on the basis of:
  - (1) the responsibilities allocate in a more natural way,
  - (2) the communication pattern between components is simpler,
  - (3) the locus of change for varied assumptions is smaller.
  - (4) (Occasionally, there appears no discernible difference between two choices. In this case, just choose one and proceed. See "Common Situations", below.)
- It is therefore often appropriate in making a well-considered decision, to interrupt a scenario on occasion and explore some variations. The variations may try alternative assumptions about future requirements or implementations, or of usage.

4. *Vary the situations, to stress test the cards*

- At any time during the walkthrough, you may vary the assumptions on the use case, to see if that causes a shift in the handling. With a good design, the handling is the same, but with the addition of a future object, or the change to at most one card.
- If it is decided a new object is needed to create a more stable design, add a new card, with the needed responsibility put onto it.
- Not all the cards on the table need be used; some may drift out to the sides if they are not used much. The cards that are needed at the end are those that get put into the design.

5. *Add cards, push cards to the side, to let the design evolve*

- CRC cards permit several design alternatives to sit on the table at the same time. An unpopular initial design may turn out to be a popular later design, or perhaps the final design is a small alteration of an initially rejected design.
- Do not throw cards away, but push them to the side, in case it turns out later they are useful.

Manage the cards.

1. *Typically, name a card immediately, but no great need to.*

- Sometimes people want to nominate a component. Fine. Put the card on the table. Ask what its responsibility is. If there appears to be a responsibility, write it down, either as an active responsibility or as a contact point. Let the card survive on its own merits.
- Sometimes people can tell that a card needs to be there, but do not know what its name or exact function is. Fine. Put a blank card there. Let its personality grow over time until its name and responsibilities become clear. Often the responsibilities will become clear first, and from the responsibilities a name will be formed.
- Sometimes a card is named, but its responsibilities evolve to a point where the name no longer matches. Draw a line through the old name and write the new. Or, get a new card, put the new name and the responsibilities on it. Put the old card to the side.

2. *Collapse cards for subsystems out of scope.*

- Frequently, the discussion goes to a different level of design. Cards are created that do not apply to the current level of design, but are useful for demonstrating the consequences of a design choice or for showing how a component would likely carry out its responsibilities.
- Rather than let the cards clutter up the table and the discussion, collect the cards that help implement a responsibility. Place them behind the component they help that is at the correct level of discussion. Then they can be brought out for examination when they are needed, and kept out of sight otherwise.
- Similarly for generic components and variants. Occasionally, the discussion will center on the generic component. The variants will be of minor importance, but are present to establish their presence. Place the variants under the generic component, so the generic component can carry the conversation. The variants can be brought out again as they are needed.

3. *Ways to collect and arrange the components:*

- **By level of implementation.** That is what has been discussed so far.
- **By privacy**. Arrange the components differently if they can be publicly known at this level of discussion, or if they are private in some way. Chances are the private components are at a different level.
- **By lifetime**. Look for components that are significantly shorter or longer lived than others. Consider whether they belong at the same level. Collecting the components by their lifetimes occasionally reveals something of interest about the system, the components, or their communications.

4. *Let unused cards drift out.*

- Of the many cards that get nominated, some do not survive through the design session. If it appears that one or more cards are not likely to see action, they may be allowed to drift to the side or back of the working area. If they develop an importance, they can be brought back into play.
- At the end of the session, if there are cards that were nominated but not used, bring them forward again for review. It should be clear that they did not manage to keep any responsibilities and so will not reach implementation. If there is disagreement on this, the person wanting to keep them must find a scenario in which they carry responsibilities.

5. *Use interaction diagrams with or even instead of cards.*

- A design group comfortable interaction diagrams may decide to let the interaction diagram carry the discussion instead of the CRC cards. This is a matter of personal preference, since some people need to see the message flows to visualize the interactions. The time view form of the interaction diagrams carries exploratory discussion better than the top view. CRC cards still offer greater flexibility and mobility in an active design session.
- If interaction diagrams are used instead of cards, write the key responsibility of each component by it on the diagram.
- At a design review, the interaction diagrams are already available as a result of the design session. The interaction diagrams may be used to illustrate how responsibilities are passed along and invoked. A listing of the responsibilities of the components must be available during the review, either as a list, or on the CRC cards, or as annotations on the interaction diagrams.
- At this point you should have a stack of cards with responsibilities, and a stack of interaction diagrams showing how the components deliver the scenarios.

Consolidate components by level.

- Identify the components that are appropriate for the level of design declared at the beginning of the design session.
- The design of the system will be presented to readers at different levels. To simplify the understanding of the design, and to isolate changes in the future, the design of the system should be documented at a consistent level. Subsystems or components that carry out the responsibilities on behalf of a component on the declared level are to be collected separately and not used in the discussion of the system at the declared level.
- At any subsequent level, the scenarios for a subsystem must have a complete description and fully connected walkthrough using only the components that are appropriate for the declared level. It is up to the design team to evaluate which components are appropriate for the level and which belong to the implementation of a component at the declared level.
- Collect separately the components and subsystems at deeper levels.
- Keep the cards for later use.
- The components outside the scope of the design are still useful. Probably, one of the design team members will be involved in the design of the subsystem using those deeper components, and will be able to use those cards to start the design.
- It is not necessary to document the use of the components at a deeper level. Someone in the room may want to document their use to help with future design or future reference..

Superclasses and Subclasses

- Superclasses and subclasses can be defined at any time they become obvious. The scenarios will illuminate these as well. It is up to the group to decide if they want to define any hierarchical relationships now or wait till the scenarios to do this.

Attributes

- Attributes of class don't really need to be defined any time soon. They are an implementation detail. The responsibilities of the class will help make these clear. Attributes are general not defined at all till the design phase, but they can be defined at anytime the group thinks it is appropriate. Remember these are implementation details and should go on the back of the card.

Scenario execution

- These are the heart of the CRC card session. Scenarios are walkthroughs of the functions of the system in detail. Take required functionality from the requirements document and use this as a scenario. Start with scenarios that are part of the systems normal operation first, and then exceptional scenarios, like error recover, later.
- First decide which class is responsible for this function. The owner of the class then picks up his card. When a card is in the air it is an object and can do things. The own announces that he needs to fulfill his responsibility. The responsibility is refined in to smaller tasks if possible. These smaller tasks can be fulfilled be the object is appropriate or they can be fulfilled be interacting with other objects. If no other appropriate class exists, maybe you need to make one. This is the fundamental procedure of the scenario execution.

B.11 REFERENCES

- The text above is adopted from the following references:
- Wilkinson, Nancy. “Using CRC Cards, An Informal Approach to Object-Oriented Development”. SIGS Publication, inc., New York, 1995.

- Nils Brummond, “Object Oriented Analysis and Design using CRC Cards”, 1998, http://www.csc.calpoly.edu/~dbutler/tutorials/winter96/crc_b/
- Alistair Cockburn, “Using CRC Cards” and “Responsibility-based Modeling”, Humans and Technology technical memo HaT TR.99.01 and TR.99.02, 1999, http://alistair.cockburn.us/crystal/articles/rbm/responsibilitybasedmodeling.html