

## CHAPTER 1: Introduction

### 1.1 Why object-oriented modeling and implementation?

In an ever changing world not only computer hardware and software are experiencing an ongoing process of innovation with more and less expensive functionality, but at the same time every human activity that uses computers is part of this “race for perfection or the best”, and faces the same challenge to constantly improve its efficiency and versatility. Modeling and the implementation of models are no exception to this trend. Both are fundamental to any discipline in natural sciences. Modeling incorporates every activity to create a model, which is a representation of a real counterpart that can be explored and manipulated to get a deeper insight into its nature. Gained knowledge can be further applied to reach pre-defined goals by e.g. creating products or services that satisfy human needs.

Examining how scientific models have evolved in time, one realizes that the first main method being used has been **functional decomposition** in conjunction with reassembling. Hereby, a problem is dismantled into its indivisible functional units for which basic behavioral rules or laws are already known, or if not, needs to be searched for. After applying these rules or laws to each unit, the final model is completed by reassembling each unit according to the conditions found before the dismantling of the considered problem. If this method of functional decomposition together with reassembling cannot be used in practice, it is at least theoretically possible, as it is done in differential and integration calculus. For example, the behavior of a multi-body system is modeled by applying the equations of motions to each body, thus using the concept of functional decomposition and differential calculus. Then the behavior of the whole system is obtained by reassembling it, that is integration over all bodies and time on the basis of their boundary conditions.

Naturally, when the age of digital computers began in the mid 1950's, the **method of functional decomposition** determined very much how the first computers were programmed. The main activity of this method concentrates on mapping the problem to a model based on functions, sub-functions, and functional interfaces. Besides, many other methods and combinations of them have evolved until now. They can be categorized into three basic groups: functional, data-

driven, and object-oriented. Their history is briefly summarized in the following table.

**Table 1.1.** Different methods for modeling and implementation

Beginning	Name	Emphasis on...	Main Concept
Mid 1950's	Functional	Functionality	Functional decomposition
Late 1970's	Data-driven	Data	Data representation
Late 1980's	Object-oriented	Behavior	Object orientation

The evolution of methods for modeling and implementation started with the functional approach in the mid 1950's using functional decomposition to create models and code mainly for processing purposes, which was the most important aspect of computer usage until the 1970's. With growing storage capabilities of computers in the late 1970's, the **data-driven method** emerged. Its main viewpoint is the flow of data and its representation by computers to find the optimum solution to questions like “How can data be most effectively stored, processed, searched, sorted, updated, etc.?” Slowly both approaches (functional and data-driven) developed into what is today commonly called **structured methodology** for modeling and implementation beginning in the early 1980's. Here **methodology** is defined as a package of methods to perform a certain problem-solving strategy, and a **strategy** consists of planned activities to reach pre-defined goals (Norman 1996). As computers grew in power, more complex problems came on the agenda, such as an interactive graphical user interface (GUI), or distributed client/server applications. Soon it was recognized that the functional and data-driven approaches are not very suitable to tackle such problems. Therefore in the late 1980's, both methodologies were basically combined to form the **object-oriented method** by creating objects that are able to encapsulate data, functionality, and behavior. After adding other concepts to this synthesis to build the main concept called “object orientation”, a more powerful methodology was born that provides more flexibility and versatility than any other approach so far.

Looking from an outside perspective, one might say that the pendulum of science history is now swinging back. In the beginning of the computer age, computer methodology was heavily borrowing from other scientific disciplines, like in the case of functional decomposition. Nowadays, ideas and techniques from computer science are moving back to other scientific disciplines to bring forth innovation. OO methodology (OOM) is one of them. It allows modeling a problem and implementing its solution more directly in a more complete “three-dimensional picture”. The question in the headline of this paragraph “**Why object-oriented modeling and implementation?**” can be briefly answered as follows where the sequence of arguments does not indicate their importance.

- OOM is using natural real-world concepts of organization (e.g. encapsulation, inheritance, polymorphism) for modeling a target problem (or system) and not something gained from a lower dimension, for example, just the processing taking place.

- OOM is a more suitable form of modularization with more self-contained and flexible units generated from a more general system concept.
- OOM offers a more natural system representation and is thus more advantageous for both client/users and designers because they allow simulating the real world more closely.
- OOM allows faster and thus more cost effective adoption to future requirements from client/users because a general concept about a system does not change as fast as more specific characteristics, allowing the reuse of components at a higher level. Therefore, the OOM focus is set more on evolutionary modeling and implementation.
- There is almost no gap between a developed object-oriented model and its implementation in an object-oriented computer program.
- OOM addresses front-end conceptual design issues, rather than back-end implementation issues. Thus, a designer can think in terms of the application domain and concentrate on the implementation until the final stages. Focusing on programming details too early restricts design choices, resulting often in less quality and more design flaws, which are more costly to fix during the implementation.
- Higher degree of modularity and flexibility, and
- Greater clarity, readability, maintainability, and reusability.

Before starting a model and its implementation, one has to decide which is the best methodology to choose for solving a considered problem. This is a very difficult decision with many factors influencing the balance of arguments. Generally, an OO methodology will pay off for complex non-linear problems, where “complex” means a problem containing many different dynamical components with variable behavior and relations. Two examples were already mentioned before: interactive graphical user interfaces and distributed client/server applications. But as long as more linear tasks with a pattern like “data input => computations => data output” need to be solved, it is better to stick to a functional or data-driven approach. It is also good to keep in mind that all methodologies (functional, data-driven, and object-oriented) overlap each other to a very high degree so that anything done with one approach can be used with another one. And there is nothing like a superior methodology. They are just different and well suited for the work they were designed for. A combination of them is also possible, but this very often lacks clarity and consistency. In the end, the final choice depends very much on the type of problem for which a solution is wanted.

## 1.2 Background of object-oriented methodology

Object-oriented methodology has emerged from various scientific disciplines like group theory, graph theory and combinatory, artificial intelligence including expert systems and neural networks, which try to describe how a more complex system operates. In its beginnings in the 1980's, the OO approach has been mainly

used for databases, expert systems, and geographical information systems (GIS). But in the last few years' object-oriented models have also been successfully applied to engineering problems like in robotics (Visinsky et al. 1994) or for the construction of machine parts (Fritzon et al. 1994, Kecskemethy & Hiller 1994). OO methodology was implemented for the first time by Kay (1969) and has brought forth since then many programming environments, among which the most famous are Smalltalk (Goldberg & Robson 1984), C++ (Stroustrup 1987) and Java (<http://java.sun.com>).

In the future, the OO approach could help us to tackle more challenging problems and model more complex systems (Coad & Yourdon 1991). A **system** consists of a number of interrelated components that work together for a common purpose. There are two types of systems: natural and fabricated (Norman 1996). Natural systems include the human body, the solar system, and the earth's climate system. Fabricated systems are created by humans to reach certain goals or serve pre-defined purposes, as automobiles, airplanes, radios, etc. are doing. In an OO perspective, a system can be defined as a collection of objects mutually interacting in the exploitation of resources and responsibilities (Baujard et al. 1994). The introduced term "**object**" can represent almost everything in our world and we can have different classes of objects. Some examples of what an object can be are material bodies, figures, physical phenomena, people, words, equations, human actions e.g. pressing mouse buttons, etc. Objects can be concrete, such as a pixel on a computer screen, or conceptual, like a priority list in a multi-user system. Generally, objects within an OO methodology are able to perceive and represent the environment in which they are placed: they may communicate with other objects and possess an autonomous behavior, depending on their resources, observations and interactions with other objects. In summary, objects know by themselves who they are, what to do, and how to behave. The OO methodology is very useful to, for example, model the behavior of complex non-linear systems, which addresses many different issues like the:

- Identification of all relevant objects,
- Description, decomposition, and allocation of their tasks,
- Interaction between objects,
- Type of communication and protocol between objects,
- Way to achieve collective behavior in order to perform organized activities and reach a common goal, or
- Choice of an appropriate implementation language and environment.

As already mentioned, one main advantage of an OO methodology is its ability to describe effectively more complex systems and to transfer the model to an object-oriented computer code without major changes, so that the gap between the designed model and the computer implementation is very small during the whole modeling process, allowing very fast design and modifications to achieve convergence. This capability allows an increasing automation, that is a simplification and acceleration of engineering model development, which is otherwise done over and

over again for slightly changed or extended problems involving a high portion of repetitive work to solve low-level problems (Fritzson et al. 1994). There seems to be a growing demand among engineers for a higher level modeling- and programming environment to realize something like “tell your computer the problem you want to solve, press a key, go home and wait for the machine to get the answer” (Hardwick & Spooner 1989; Sanal 1994). Object-oriented techniques seem to be more advanced and suitable tools for client/users and designers to come closer to such a dream.

### 1.3 Key concepts of object-oriented methodology

Object-oriented methodology seeks to mimic the way we form models of the world. To cope with complexities of life, we have evolved a wonderful capacity to generalize, classify and generate abstractions. Almost every noun in our vocabulary represents a class of objects sharing some set of attributes or behavioral traits. OO methodology exploits this natural tendency we have to classify and abstract things. Object-oriented methodology is implemented by object-oriented programming. Supporting and enforcing the use of the following key concepts whereby in an OO terminology, “data” generally corresponds to “attributes”, and “functions” are called “methods” or “operations”, can characterize both of them. To help the reader to adapt, these OO terms are already used in the following list of **key object-oriented concepts**.

- **Encapsulation** is the combining of attributes with the operations dedicated to manipulate the attributes. Encapsulation allows the hiding of information and is achieved by means of a new structuring and data-type mechanism, which is named “class”.
- **Inheritance** is a concept for expressing similarity by which a newly built, derived class inherits the attributes and operations from one or more ancestor classes, while possibly redefining or adding new attributes and operations. This creates a hierarchy of classes like an inheritance tree.
- **Polymorphism** is a concept for changing the behavior of operations when used by different classes. Or in other words, a operation with a given name is shared up and down the class hierarchy, with each class in the hierarchy implementing the operation in a way appropriate to itself.
- **Dynamic (or late) binding** is the concept to implement polymorphism by which the physical behavior of an object is prolonged to the adequate moment at run-time of the program.

Encapsulation greatly facilitates the usage of complex data structures to model a complex system, thus allowing for the collection of information of different types that belong naturally together in one class as a single unit, e.g. data of a finite element node containing its number, coordinates, type and value of boundary condition (Forte et al. 1990). Therefore, encapsulation increases the readability and maintainability of the model and code (Sanal 1994). Further, encapsulation

sets a boundary between a class and the rest of a program, hiding and protecting its contents from unwanted side effects and thereby making classes more robust. This characteristic of encapsulation is commonly called “information hiding”, allowing increasing the modularity of model and code (Norman 1996).

It is also very beneficial to use existing classes when defining new classes. This can be achieved by composition, or by extension of already defined classes using the concept of inheritance. The main purpose of inheritance is to allow sharing of attributes and operations among classes according to a hierarchical relationship, allowing simplifying model and code. From one class, any desired number of class objects can be initialized and used, a process similar to the casting of pieces from a single mould. Each of these copies is autonomous, doing its job on its own, but if it needs some data or code from another class object, the concept of inheritance allows them to share it. Inheritance is of crucial importance for the realization of tree-structured dependencies between class objects. The ability to transfer common attributes and operations of several classes to a base class and inherit them can greatly reduce complexity and repetition within a model. This is one major benefit of the object-oriented approach.

**Table 1.2.** List of object-oriented terms

Name	Definition / Meaning
Object	An instance of a particular class
Class	A template for objects to encapsulate data, functionality, and behavior
Member	A variable (attribute) or function (operation, service, procedure) within an object
Encapsulation	The binding of attributes and operations into a class of objects
Inheritance	Means that derived classes inherit the attributes and operations from their ancestor classes, while possibly redefining or adding new attributes and operations. This creates a hierarchy of ancestor classes.
Polymorphism	The ability of classes in a hierarchy to share an operation with the same name behaving appropriately to the particular class calling the operation
Late binding	The binding of virtual functions to an object when it is created during run-time
Constructor / Destructor	Operations to create / eliminate objects

Sometimes, a set of class objects may share an operation, but it can only be defined at a specific program level or stage. Thus, this operation can be initialized as a virtual function, but later defined at run-time when it is required. This process is called dynamic (or late) binding. It is part of the polymorphism concept, which makes it possible to adopt an initialized operation to the needs of an object class in the same inheritance tree by adding new features or overwriting existing ones.

Polymorphism simply means that an operation with the same name is able to behave differently in different classes. All relevant object-oriented terms are summarized with their meaning in Table 1.2 (McMonnies & McSporran 1995).

All four concepts are the foundation of any object-oriented methodology, which is a new way of looking at, analyzing, modeling, designing, and implementing models, concentrating on the underlying real-world concepts and principles of a problem or system, rather than processing or implementation details. Thus, there is a general trend in, for example, software development away from programming language issues towards the fundamentals and rules within the application domain, which includes the identification and organization of system components and concepts, their behavior, hierarchy, and relations. This is of great importance for increasing the productivity of modeling and coding as well as reducing the software mountain because general concepts enjoy a longer life span than some implementation details.

In conclusion, object-oriented methodology is a conceptual approach independent from a programming language until the last stages (Rumbaugh et al. 1991). It is a fundamental new way of thinking about and modeling a problem or system, but not a programming technique. Its greatest advantages originate from its capability to create a better and more complete picture of a problem domain. It serves as a more efficient platform for all parts of the modeling and implementation process including concepts, specifications, analysis, design, implementation, testing, documentation, distribution and maintenance.

## 1.4 Breadth-first approach to OO methodology

The breadth-first approach adopted in this chapter for beginning to teach object-oriented methodology gives exposure to just the essential concepts and elements of OO methodology, but relegates depth of knowledge to later chapters in this book (Nagin & Impagliazzo 1995). Thus, the reader should be able to:

- Understand how an OO methodology works in principle,
- Realize the simplicity and elegance of an OO methodology, and
- Realize the potential and advantages of an OO methodology.

An interesting and scientifically relevant example was chosen to take the reader through all stages of an OO modeling and implementation, starting with the first idea and ending with the final program (or parts of it). The example deals with a second boundary value problem of an accelerated projectile to demonstrate, for example, the benefits of computational experimentation. Naturally, the chosen example cannot be too long and complicated, ensuring an easy-to-follow writing style, which is a major goal of this book.

### 1.4.1 Basic idea and specifications of an OO example

The basic idea and specifications of the example are the following ones. A basketball game between two teams, the “A-City Lions” against the “B-City Tigers”, should be simulated by a computer, whereby the *main (or parent) window* of the screen is equally divided into two parts called *child windows* corresponding to two halves of the field with the baskets of each team at both ends. Alternately, players should be positioned by moving the mouse and clicking its buttons, and in a similar way, the speed and direction of a shot should be selected. Then the ball orbit should be immediately drawn, the results displayed and automatically counted. To make it simpler, the problem is reduced to two dimensions in a side view perspective, and certain elements are kept constant like the position and size of the basketball and the field. In summary, the task is to create an interactive window program that takes advantage of its built-in multi-tasking and point-and-click facilities to simulate a basketball game and illustrate the second boundary value problem of an accelerated projectile.

### 1.4.2 Starting the analysis: what concepts and/or laws are governing the considered problem?

Historically and still today, the accelerated motion of a projectile (here a basketball), beginning at a given starting position (here a basketball player) and ending at a fixed target position (here a basket), is a research topic not only in sports (a little joke), but also in astronomy, geodesy, or ballistics (Falk & Ruppel 1983). Further, it is one of the crucial concepts to grasp in **classical mechanics** and therefore, student instruction should pay adequate attention to it. Mathematically, it is a second boundary value (inverse) problem to Newton's second law of motion, which is an ordinary differential equation of second order in time. If this law is reduced to a two-body problem with only central symmetrical gravity fields, an analytical solution can be obtained in form of Kepler's laws (Heitz 1980). Generally, the second boundary value problem can be solved by using the first one where all starting values are given in one location at the same time in combination with a predictor-corrector technique. This is exactly what basketball players are doing: they perform experiments based on a trial-and-error method by changing their starting values (player position, speed and direction of the ball when shot) in such a way that the ball flies into the basket. For simplicity, only the two-body problem together with a numerical integration scheme is employed, but to make the game more interesting and increase its learning effect, a team should be able to change the position and value of the central gravity mass in its own half of the field within reasonable boundaries after the other team has scored. Thus, the simulated game should use two different gravity fields in one geographical location, including an element of fiction in order to visualize the effects of changing physical parameters and experience alternative realities. This allows the user to experience the consequences of breaking physical laws, providing opportunities for comparative testing of different models and comprehension of their underlying logic (Hennessy et al.



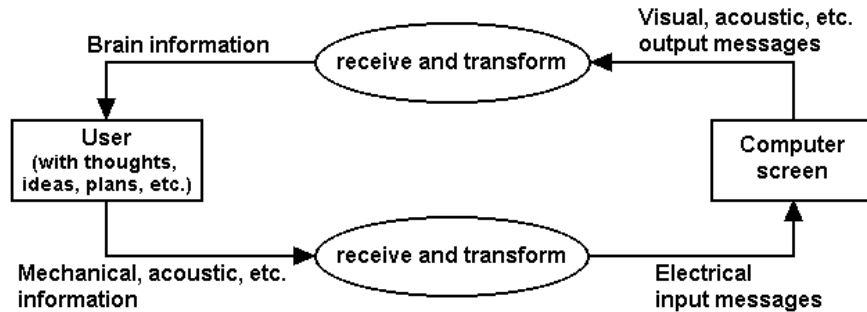
1995). Such a program could also serve as a **computer-assisted learning** (CAL) tool, which can be of great benefit in science education where many students have learning difficulties in acquiring knowledge about fundamental mechanical concepts like gravity, mass or acceleration (McDermott 1990, 1991). Moreover, the OO methodology applied here allows an easy incorporation of other forces such as friction or inertia forces.

### 1.4.3 OO modeling of a basketball game simulation

The method of creating the object-oriented model for simulating the specified basketball game is the object modeling technique (OMT) proposed by Rumbaugh et al. (1991) which allows an object-oriented analysis of complex systems and builds naturally on the key OO concepts introduced before. The OMT was selected for this breadth-first approach because it is the simplest OO approach to learn in the author's experience, partly due to strong similarities to the structured approach (Sully 1993). But instead of just modeling the data flow as done in the structured method, the OMT uses different projections of varying complexity for the same system like in modern computer drawing programs in order to model the system's information, behavior, and functionality. All projections named **models** depend on each other (McMonnies & McSporran 1995): an object model, which represents the static framework of the system (i.e. what the components are and how they relate to each other) expressed by class and object diagrams; a dynamic model, which describes the events, states, and conditions influencing the system illustrated by state diagrams; and a functional model, which represents the data transformations of the system illustrated by data flow diagrams (where data comes from, what it is transformed into, and where it goes to). As technical drawings are using three different viewpoints (side, front, and top) of the same item, the three models of the OMT are orthogonal parts of one description to represent the whole system. A nice feature of OMT is that the same diagrams and notation can be used for all modeling and implementation stages.

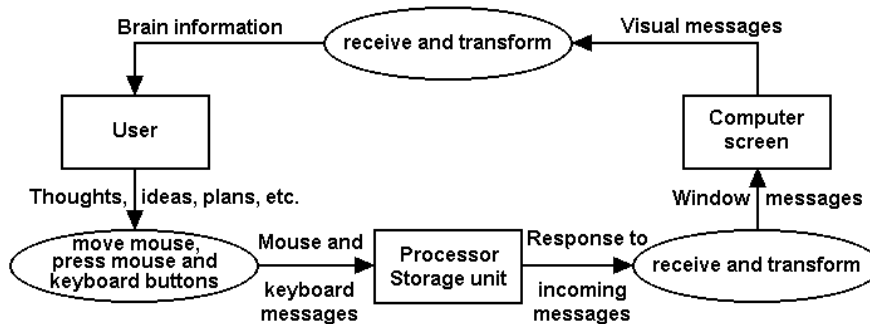
#### 1.4.3.1 Definition of the system domain and boundaries

Before developing the object-oriented model a brief look at the system's functional model gives an idea about the system domain and boundaries, allowing their appropriate definition. Starting with the physical components of the system, a first simple functional model of human interaction with computers is shown in Fig. 1.1.



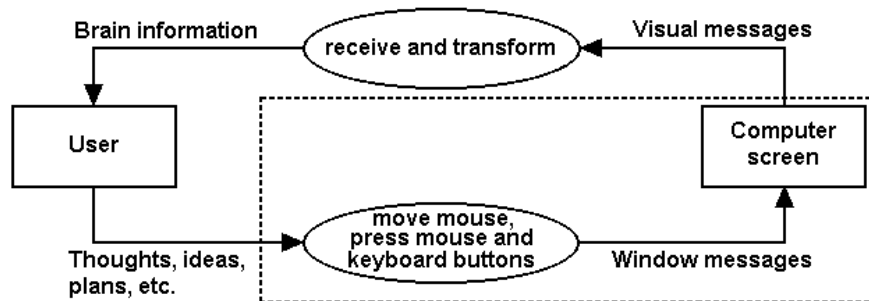
**Fig. 1.1.** Counter-clockwise data flow diagram for the interaction between user and computer

A user sends mechanical, acoustic or other information based on thoughts, ideas, plans, etc. to a computer which receives and transforms them into electrical input messages. After reacting accordingly the transmittal is reversed; the user receives visual, acoustic, etc. output messages from the computer and transforms them, by using human senses, into brain information. Concentrating only on mechanical and visual means relevant to the application considered here, Fig. 1.1 can be better specified as follows.



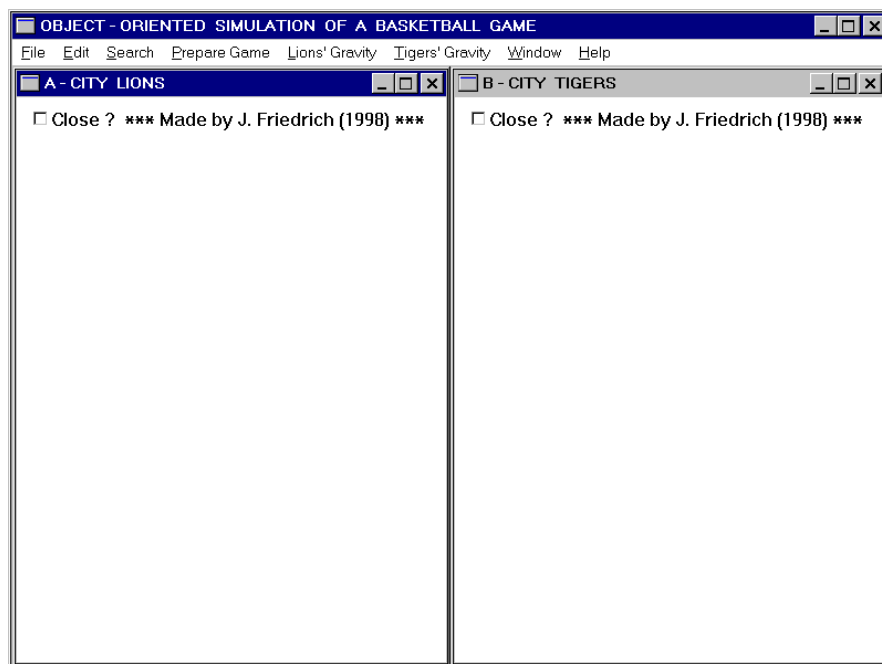
**Fig. 1.2.** More detailed data flow diagram for the interaction between user and computer

To keep things as clear as possible, the system is reduced to what a user is physically doing: watching a computer screen and using mouse / keyboard as input devices so that Fig. 1.2 simplifies to Fig. 1.3.



**Fig. 1.3.** Counter-clockwise data flow diagram with the system domain and boundary as dotted line

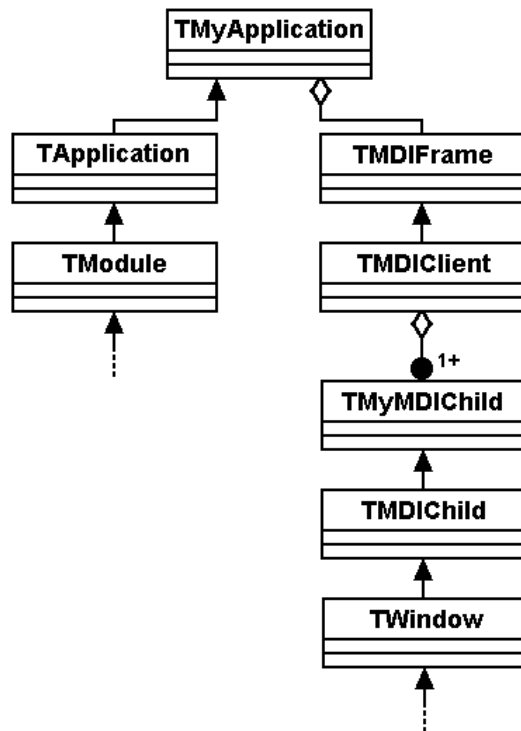
The dotted line in Fig. 1.3 marks the boundary of the system domain. Thus, the only directly used physical system components are mouse, keyboard, and computer screen.



**Fig. 1.4.** Window display of the OO simulation of a basketball game

### 1.4.3.2 Object model

The object model identifies and describes the system components and their structure. The system domain is the display of a window program with a main (or parent) window and, for multi-tasking, several child windows (here two are needed), each containing basic window elements like a title bar and control boxes (Fig. 1.4). Moreover, the main window in Fig. 1.4 contains a menu bar for standard window (File, Edit, Search, Window, Help) and application-specific functions (Prepare Game, Lions' Gravity, Tigers' Gravity). Such a screen can be created using window libraries like those provided with C++ compilers for the Microsoft Windows<sup>TM</sup> operating system, for example, the Borland C++ compiler (1994) used for this program. Thus, classes of this compiler like *TApplication*, *TMDIFrame*, *TMDIClient*, *TMDIChild* and *TWindow* serve as base classes for self-written (by the programmer) derived classes needed for the application-specific functions completing the class diagram shown in Fig. 1.5 (Class names in the text are written in italics; classes containing *My* in their name are self-written; other classes are taken from the compiler's window class library).



**Fig. 1.5.** Class diagram of the OO simulation of a basketball game

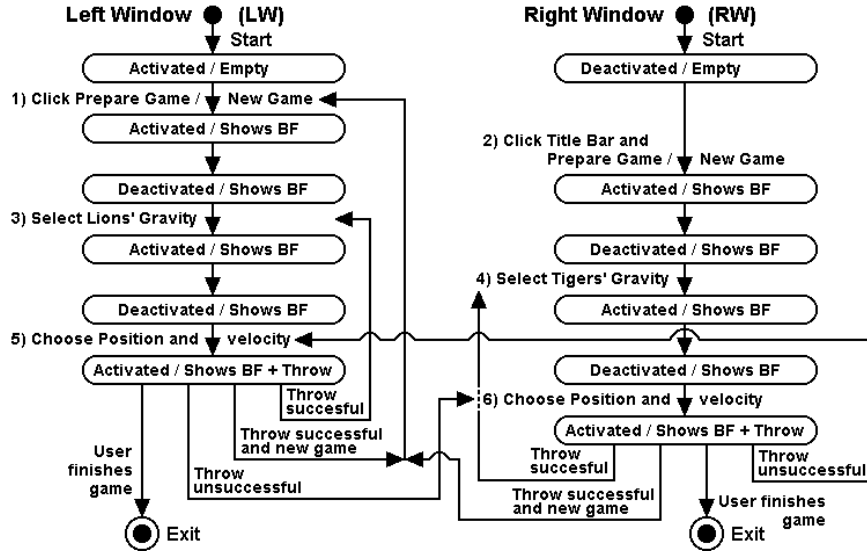
Fig. 1.5 displays only a very basic object model with the most important classes. The complete model of the final program is much more detailed and can be viewed by using e.g. the class browser of the used compiler. It includes, for example, classes to create and handle file menus and dialog boxes.

*TMyApplication* is derived from *TApplication* and *TModule* (indicated by an arrow for a one-to-one relationship), both responsible for the basic behavior of a window program including their initialization and message processing. A constructor of *TMDIFrame* sets up the main window or frame of *TMyApplication*, which is a part of *TMyApplication* (indicated by the diamond symbol). *TMDIClient* manages the child windows inside of the frame. All *MDI* classes belong to the multiple-document interface (MDI) that takes care of e.g. constructing and handling parent and child windows. *TMDIClient* creates child windows by calling a *TMyMDIChild* constructor. Thus, a one-to-many relationship exists between *TMDIClient* and *TMyMDIChild* (indicated by the diamond and circle symbol), a class derived from *TMDIChild* originating from *TWindow*. The *TWindow* class provides all the window-specific functionality for a child window. *TWindow* and *TModule* build upon other window library classes not mentioned here (for further information see Borland (1994)). In this example *TMyMDIChild* is of special importance because it supplies everything needed for the target model and code.

### 1.4.3.3 Dynamic model

The system domain in Fig. 1.4 consists of a main window including menu bar items and two child windows as system elements. The feature of a window program which makes it so effective is its ability to change the state (activated or deactivated) of any system element by just positioning the mouse on a considered element and pressing a mouse button (normally the left one), which is the event causing the element's state to change. This is automatically performed by virtual event message response functions of the built-in window library classes (Borland 1994).

The execution of menu bar items or one of their corresponding pop-up choices takes place in a similar way. If the item is provided by the compiler's window class library like for File, Edit, Search, Window, or Help, the response happens accordingly to the built-in library functions. The Help item is placed into this category because only its contents need to be adjusted to the present application. A self-created item such as Prepare Game, Lions' Gravity, or Tigers' Gravity is linked to a virtual function of *TMyMDIChild* through one command message (CM) identifier given to both the menu bar item and the corresponding member function when they are declared. Virtual window message (WM) response functions and identifiers process analogously mouse moves and presses inside of an active child window. Both CM and WM functions are essential tools of the designed dynamic model shown in the state diagram in Fig. 1.6.



**Fig. 1.6.** State diagram of the OO simulation of a basketball game

At the beginning of the game after starting the program, the left window (LW) in Fig. 1.4 entitled “A-CITY LIONS” is activated shown by its highlighted title bar, and the right window (RW) for the other team “B-CITY TIGERS” is deactivated. Both windows are empty. The game begins after the menu bar item Prepare Game / New Game is clicked in both windows, which sets the score counters to zero and displays the basketball field (BF) (step 1 and 2). Step 3 and 4 allow both teams to choose their gravity location by selecting the menu bar items Lions' Gravity and Tigers' Gravity. Then either team can start the game, for example, the Tigers. They move the mouse to the Lions' LW, press the left mouse button down (only this button is used throughout the whole program) at the wanted position of their player and keep the button down until the mouse is moved to and released at a second position (step 5). The difference between the first and second position  $P_i^1 - P_i^2$ ,  $i \in \{x, y\}$ , defines the ball's velocity vector used to compute its orbit, immediately shown in the LW after releasing the mouse button. Then the Lions start their game by moving the mouse to the Tigers' RW, following the actions of the 5<sup>th</sup> step in an analogous way (step 6). Steps 5 and 6 are repeated alternately until one team scores, so that the score counter at the top of each window is incremented by two (Fig. 1.8). In order to make the next attempt harder, the other team is allowed to change its gravity location and value (step 3 or 4). The game is finished after reaching a score or time limit. Going back to the first step can start a new game.

#### 1.4.3.4 Functional model

The functional model completes the object and dynamic model by concentrating on the flow of data between stores and processes. Processes are all operations that transform data inputs into data outputs. Employing the sequential steps of Fig. 1.6, the simplified data flow diagram in Fig. 1.3 can be specified as shown in Fig. 1.7. The data is only generated by the ideas, plans, and decisions of the users (the two teams Lions and Tigers) wanting to perform the visual interactive simulation of a basketball game by adequate processes (mouse moves and mouse/keyboard button presses) according to the object model explained and shown before in Fig. 1.6.

Therefore, all data comes from the users (1<sup>st</sup> data store) and flows to the computer screen (2<sup>nd</sup> data store) where it is stored as visual data as screen contents. There it lasts and remains visible until it is changed by the window display as response to later commands, for example, a newer one replaces the old graphics of a basketball field.

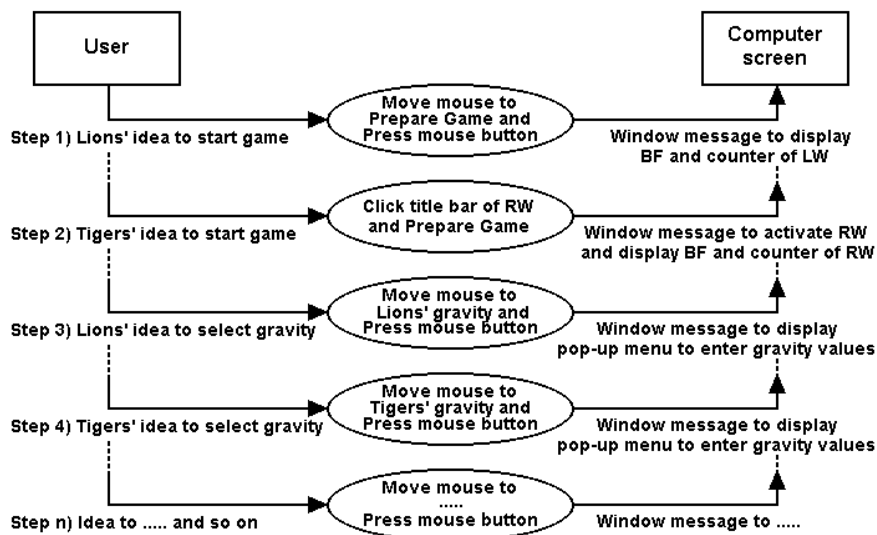


Fig. 1.7. Data flow diagram of the OO simulation of a basketball game

#### 1.4.4 OO implementation of a basketball game simulation

For implementing the created model, a programming language has to be selected, which always includes a compromise because there is nothing like a “perfect” programming language yet. Here C++ was chosen because:

- It is a multi-purpose language capable of solving both high and low-level computer problems.
- It offers together with ANSI C a large selection of existing libraries and other resources (e.g. text books) for many different applications, e.g. for numerics, databases and graphics (e.g. OpenGL).
- It runs on almost every hardware and operating system, and
- Offers OO facilities, such as dynamic link libraries (DLLs) or CASE tools.

The implementation of virtual CM and WM response functions, which represent the largest portion in the final program, are just briefly explained in this paragraph and illustrated by the following C++ code (for further information see e.g. Borland (1994) or Heiny (1994)).

**Table 1.3.** Example C++ code for virtual CM and WM response functions

01	...	// Declaration of class TMyMDIChild
02	class TMyMDIChild : public TWindow { public:	
03	TMyMDIChild (TWindow &parent, const char far *title);	// Constructor
04	~TMyMDIChild ( );	// Destructor
05	int x,y, vx,vy;	// Four integer numbers for position and velocity
06	...	// Declaration of virtual functions:
07	virtual void EvLButtonDown (UNIT, TPoint&);	// Left mouse button down
08	virtual void EvLButtonUp (UNIT, TPoint&);	// Left mouse button up
09	DECLARE_RESPONSE_TABLE (TMyMDIChild);	
10	};	//Declare response table
11	DEFINE_RESPONSE_TABLE1(TMyMDIChild, TWindow)	
12	EV_WM_LBUTTONDOWN,	// Definition of response table
13	EV_WM_LBUTTONUP,	
14	END_RESPONSE_TABLE;	
15	...	// Definition of virtual functions
16	void TMyMDIChild::EvLButtonDown (UINT, TPoint& Coord) {	
17	HDC hdc = GetDC (Hwindow);	// Get window device handle
18	x = Coord.x; y = Coord.y;	// Store window coordinates of mouse in x, y
19	ReleaseDC (Hwindow, hdc);	// Release window device
20	};	
21	void TMyMDIChild::EvLButtonUp (UINT, TPoint& Coord) {	
22	HDC hdc = GetDC (Hwindow);	// Get window device handle
23	Vx = x - Coord.x; vy = y - Coord.y;	// Store velocity components in vx, vy
24	ReleaseDC (Hwindow, hdc);	// Release window device
25	};	

Here, pressing the left mouse button down at a first position stores the mouse coordinates within the activated window in the integer variables x and y. Releasing the same button at a second position computes the ball's velocity components vx and vy as the difference between the first and second mouse position  $P_1^1 - P_1^2$ ,



$i \in \{x, y\}$ . Both virtual functions “EvLButton-Down” and “EvLButtonUp” of *TMyMDIChild* respond to the incoming window messages `EV_WM_LBUTTONDOWN` and `EV_WM_LBUTTONUP` defined in the response table of *TMyMDIChild*. “Coord” is an instance of class *TPoint*, another built-in window class of the compiler. It contains the mouse coordinates of the last event when a mouse button was pressed. “UINT” is an identifier for combined keyboard and mouse events like “Shift + Double-Click”, but here not used. A typical screen contents after starting a game looks as is shown in Fig. 1.8 (Download latest version at <http://www.simtel.net/pub/pd/53592.html>).

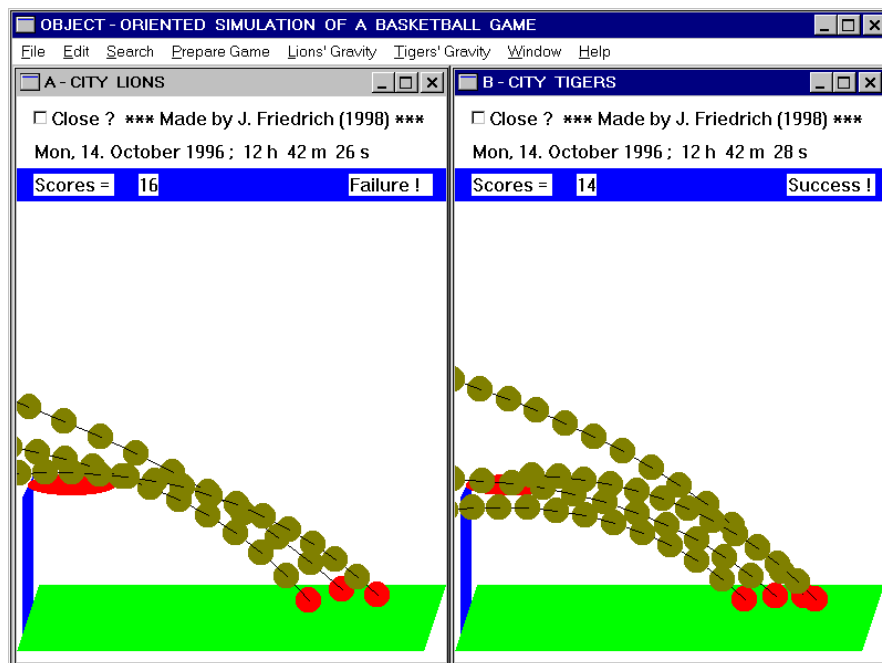


Fig. 1.8. Screen shots after starting the OO simulation of a basketball game

As this OO simulation program of a basketball game illustrates, a window program is organized around events and messages within an object-oriented programming environment. All window interface elements like child windows, scroll bars or dialog boxes are objects that combine data with functionality. For example, pressing the OK button in a dialog box may be connected to a virtual function that verifies the entered data and passes them on to some other variables. Such a style of organization can be modeled by, for example, the **responsibility-driven approach** of Wirfs-Brock et al. (1990). The authors have basically divided all involved objects into clients and servers, such that the servers provide some services laid down in a server-specific contract, which can be invoked by clients, who do not have to know how the services are carried out. The main characteristic of this client/server model is the emphasis on actions within object relations instead of

representations. The object-oriented model for the basketball simulation program utilizes this responsibility-driven approach and divides all objects into two categories: **state objects**, which are able to hold information, and **acting objects**, which receive this information from a set of state objects, perform their task, and transfer all their knowledge back to the same or another set of state objects, or, if necessary, to some other type of objects. Thus, state objects play the role of clients, acting objects the role of servers, which can be also regarded as mappings between two sets of state objects. Note that the concept of treating state- and acting objects as separate, autonomous entities is very similar to the understanding of nodes and elements in boundary or finite element theory (Brebbia 1978, Zienkiewicz 1977). But the difference is that the two categories of objects introduced here are made visible and approachable, allowing the designer to use them as building blocks of a “**unit construction system**” for modeling other interactive multi-object applications, thus increasing reusability and maintainability.

Such an object-oriented modeling and implementation style benefits very much from the built-in window libraries. For example, both the C++ “Object Windows Library” (OWL) from Borland or the “Microsoft Foundation Class” (MFC) contain over one thousand functions, thus indicating the great potential of such applications. Two other benefits of the introduced approach are the capability of parallel processing on personal computers (PCs) by using multi-threading (a *thread* is a block of code in a computer program), and the design of faster 32-bit programs for PCs.

Regarding the implementation of object-oriented models, another new programming language called Java<sup>TM</sup> by Sun Microsystems Inc. already became so popular and widely used since its introduction in 1995 that the amount of OO models implemented in Java is sharply increasing. This book introduces two Java applets at the end of part II: a city map applet with a road finder in chapter 9 and in chapter 10, an applet for parallel computing with Java threads (also called “multi-threading”) to determine orbits of earth satellites.

Spatial Modeling in Natural Sciences and Engineering  
Software Development and Implementation

Friedrich, J.

2004, XV, 305 p. 36 illus. in color., Hardcover

ISBN: 978-3-540-20877-8