DIFFERENCES BETWEEN THE STRUCTURE OF OBJECT-ORIENTED PROGRAMS AND HYPERTEXTS

Diploma Work

Kamil Drobný
March 1998
Declaration

I honorably declare that I have done this work independently using only the listed literature.

Bratislava, March 98
Acknowledgments

I would like to give special thanks to God, because he loves all the people (especially me) and to my tutor Dr. Veith Risak for his endless stocks of valuable advices and for his readiness to always help me and to Marek and Jonatán for their comments.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>1</td>
</tr>
<tr>
<td>1 Preface</td>
<td>3</td>
</tr>
<tr>
<td>1.1 Hypertext and object-oriented programming</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Overview of hypertext history</td>
<td>3</td>
</tr>
<tr>
<td>2 Introduction to hypertext</td>
<td>6</td>
</tr>
<tr>
<td>2.1 What is hypertext</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Hypertext terminology</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Advantages of hypertext</td>
<td>9</td>
</tr>
<tr>
<td>2.4 Problems in hypertext</td>
<td>9</td>
</tr>
<tr>
<td>2.4.1 Syntactic errors</td>
<td>10</td>
</tr>
<tr>
<td>2.4.2 Semantic errors</td>
<td>11</td>
</tr>
<tr>
<td>2.5 Quality factors</td>
<td>13</td>
</tr>
<tr>
<td>2.5.1 External factors of programs quality</td>
<td>13</td>
</tr>
<tr>
<td>2.5.2 External factors of the quality from the hypertext view</td>
<td>14</td>
</tr>
<tr>
<td>3 Summary of OOP principles</td>
<td>17</td>
</tr>
<tr>
<td>3.1 Conception of object</td>
<td>17</td>
</tr>
<tr>
<td>3.2 Terminology of object-oriented programming</td>
<td>17</td>
</tr>
<tr>
<td>3.3 Object data management and requirements</td>
<td>22</td>
</tr>
<tr>
<td>4 Description of hypertext systems and models, looking for object-oriented characteristics</td>
<td>27</td>
</tr>
<tr>
<td>4.1 Introduction to hypertext systems and models</td>
<td>27</td>
</tr>
<tr>
<td>4.2 Description of hypertext systems</td>
<td>31</td>
</tr>
<tr>
<td>4.2.1 Dexter Hypertext Reference Model</td>
<td>31</td>
</tr>
<tr>
<td>4.2.2 LANGE Formal Model of Hypertext</td>
<td>35</td>
</tr>
<tr>
<td>4.2.3 Hypertext Abstract Machine</td>
<td>36</td>
</tr>
<tr>
<td>4.2.4 HM Model</td>
<td>37</td>
</tr>
<tr>
<td>4.3 Description of hypertext systems</td>
<td>39</td>
</tr>
<tr>
<td>4.3.1 KMS</td>
<td>39</td>
</tr>
<tr>
<td>4.3.2 Guide</td>
<td>39</td>
</tr>
<tr>
<td>4.3.3 HyperCard</td>
<td>41</td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>----</td>
</tr>
<tr>
<td>4.3.4 Intermedia 3.0</td>
<td>43</td>
</tr>
<tr>
<td>4.3.5 HTML</td>
<td>43</td>
</tr>
<tr>
<td>4.3.6 ToolBook</td>
<td>46</td>
</tr>
<tr>
<td>4.3.7 HyperTies</td>
<td>46</td>
</tr>
<tr>
<td>4.4 Summary of object-oriented characteristics</td>
<td>47</td>
</tr>
<tr>
<td>5 Consideration of possibilities of using o-o features in development of</td>
<td></td>
</tr>
<tr>
<td>hypertexts and proposal of class hierarchies for links and nodes in hypertext</td>
<td></td>
</tr>
<tr>
<td>with o-o features</td>
<td>49</td>
</tr>
<tr>
<td>5.1 Object-oriented hypertext model</td>
<td>49</td>
</tr>
<tr>
<td>5.2 Data and schema</td>
<td>55</td>
</tr>
<tr>
<td>5.3 Creation of schema in object-oriented hypertext model</td>
<td>58</td>
</tr>
<tr>
<td>5.4 Encapsulation - the main characteristic of the object-oriented approach</td>
<td>64</td>
</tr>
<tr>
<td>5.5 Links as independent objects</td>
<td>66</td>
</tr>
<tr>
<td>5.6 Utilization of the other object-oriented features</td>
<td>69</td>
</tr>
<tr>
<td>6 Summary</td>
<td>71</td>
</tr>
<tr>
<td>6.1 Main results</td>
<td>71</td>
</tr>
<tr>
<td>6.2 Quality factors of object-oriented hypertext systems</td>
<td>71</td>
</tr>
<tr>
<td>7 Literature</td>
<td>74</td>
</tr>
<tr>
<td>7.1 Literature in paper form</td>
<td>74</td>
</tr>
<tr>
<td>7.2 Electronic documents</td>
<td>75</td>
</tr>
<tr>
<td>8 Appendix</td>
<td>76</td>
</tr>
</tbody>
</table>
1 Preface

1.1 Hypertext and object-oriented programming

Nowadays we can find hypertext systems in many computer applications. Hypertext is not only Internet and HTML. Every new OS, programming language, text editor, application program, game or file manager uses hypertext in some way. They use hypertext especially since the arrival of WWW and they use it for helps and tutorials, but we can find it in many other forms too. Hypertext becomes very popular. More and more information is available in form of hypertext. This means, that we create more and bigger hypertexts, which leads to more and bigger problems. We also want to know how to make hypertexts faster, more reliable and how to avoid problems, which come with the development of big hypertext systems.

We have many experiences from the development of big programs and we found some solutions for efficient programming of big and reliable programs. Therefore, it is natural, that we try to use these solutions to solve similar hypertext problems. One of the possibilities how to do a "good" program is object-oriented programming.

This diploma-thesis wants to consider possibilities to use the object-oriented approach for a development of hypertext.

This work can be divided into three parts. In the first part basic conceptions of hypertext and object-oriented approach are explained. The second one describes some existing hypertext systems and models and shows their object-oriented characteristics, if some of them was found. The main part of this diploma-work is the third one. It contains proposal for an object-oriented hypertext model and discusses possible advantages of implementations of this model.

1.2 Overview of hypertext history

One of the first, and to date most articulate proponents of what we call now hypermedia, was Vannevar Bush [Bush-45], senior science advisor to President Roosevelt, and administrator of the wartime Manhattan Project. His proposal for a Memex device is designed around the concept of association - in both the traditional philosophical and more recent psycho-physiological senses. He describes the Memex as
"a sort of mechanized private file and library". This device should support the selection of information by association rather than by indexing. Users would be able to record, annotate items, and link them by trails, which are then named and stored.

Early hypertext systems were produced often 1960s. They were primitive and experimental. They had many limitations of the size of nodes and the amount of information to be displayed at one time; they used the keyboard as the only available input device, they could display text characters only, but not graphics. Among the pioneers of this period was Douglas Engelbart [Engelbart-63] and Ted Nelson [Nelson-81]. Nelson's ideas revolved around the system called "Xanadu" in which a user could create an "hypertext", a document consisting of linked "nodes". Nelson called it "non-sequential writing".

Engelbart developed the NLS (On-line System) and later Augment systems. These are based around a hierarchical text-fragment database, mediated by selective "view filters" and display "views". These filters allow hierarchy-based operations such as level clipping and truncation, and could be extended using the system's own high-level description language. Among the innovative features of NLS were its multi-user capability and multiple-windowing, as well as two peripheral control devices to be used with the keyboard: short keys and the mouse. Both Bush and Engelbart believed in the ability of a machine to enhance man's intelligence, and to communicate that intelligence; the emphasis, however, is on personal information, which is stored locally.

One of the best-known hypertext systems of the 1970s was ZOG, a frame-based text database developed at Carnegie-Mellon University. Hierarchically linked, and ZOG's screen-sized frames of information and menu options provided a consistent interface. The commercial version, KMS (Knowledge Management System) incorporated graphics facilities along with text. This ability to handle graphics as well as textual notes was fundamental in Vannevar Bush's model of the thinking machine, as reflecting the usual means which people use to store and communicate information.

Among the other early hypertext systems, which have been augmented with graphics facilities are Hyperties and Intermedia. The trend was much boosted by the introduction of high-resolution screen displays, and the mass production of memory chips capable of driving them. One of the earliest microcomputer implementations of hypermedia was
Guide. However, the most significant break-through in terms of popularizing hypermedia came with Apple Computer’s decision to bundle a program called HyperCard with new Macintosh machines. HyperCard offered users the ability to link text and graphics, to multi-task between other applications, and for the first time, a user-friendly, high-level programming environment was available for the Macintosh - with HyperTalk.

A HyperCard combines aspect of relational database models with the ability to integrate materials from heterogeneous external sources such as digitized audio and video. It capitalizes on the graphical Macintosh interface: the use of mouse-driven menus, buttons and icons, as well as providing conceptually simple development tools.
2 Introduction to hypertext

2.1 What is hypertext

When we look to the Oxford dictionary of computing [Oxford-96], we can find this definition of hypertext: "text stored in a computer system that contains links that allow the user to move from one piece of text to another". From another view, hypertext is a way to present information. The next definition says that hypertext is a non-linear form of writing. However, when we want to have a more abstract definition of hypertext, we should imagine a graph, where nodes are pieces of texts or other information and the edges are links among them. Example of an hypertext and its graph is in Figure 1 and Figure 2.

![Figure 1](image1.png)

**Figure 1.** Example of an hypertext.

![Figure 2](image2.png)

**Figure 2.** Graph represented previous hypertext example.

When we read an hypertext, we are always in one of the nodes. If we need more or other information we use one of the references to other nodes and the browser makes a "jump" to this node.
2.2 Hypertext terminology

This chapter serves as an introduction to the hypertext terminology. Here is a list of the basic conception:

- **Anchor**
  
is the source of the link. It means, that the link begins here. It is some area in the node, usually highlighted, underlined or emphasized in some other proper way. By placing the cursor on the anchor, we can activate the link beginning at this anchor and go to the other end (target) of this link. In some models anchor also means the position in the target node where the link points.

- **Author**
  
a writer of hypertext. He creates not only texts in nodes, but also connections among them. Therefore, he has the responsibility for the whole hypertext.

- **Author mode**
  
mode of the browser or hypertext editor, where the user can work with the hypertext document as an author. He can create, move, modify, view or delete links or nodes.

- **Authoring**
  
a process of creating or modifying hypertext.

- **Back link**
  
connection to the node I came from. The majority of hypertext systems enable back linking.

- **Browser**
  
program for reading hypertext. Browsers allow us to see one node of hypertext and jump to the linked nodes. Readers use the browser to navigate in reading hypertext. Modern browsers support many multimedia features.

- **Card**
  
name for a node in some hypertext systems. This involves some restrictions for a node. A card has a fixed constant length; links can be targeted only to the beginning of the cards. A card can be usually placed at one single screen.
• **Dangling link**
  
  a link without a source or target. Dangling links are quite common in hypertexts under construction. In finished hypertexts the appearance of dangling link means an error and if a reader tries to use such a link, it usually leads to an error message (HTML) or link stays inactive, invisible (HyperTies).

• **HTML**
  
  HyperText Mark-up Language, the language for writing hypertexts used in the World Wide Web hypertext system.

• **Hypermedia**
  
  hypertext with multimedia extensions. It works not only with text, but also with sound, pictures, graphics, video-sequences and so on.

• **Hypertext**
  
  a non-linear text distributed among nodes, which are connected by links. Sometimes the term hypertext means both hypertext and hypertext-system.

• **Hypertext-system**
  
  hardware and software, which allows the users to use the hypertext and to gives them access to information from the hypertext. Tools for creating hypertexts are also part of hypertext systems.

• **Link**
  
  a connection between two parts of the text (nodes). A link is created using anchors and can lead from one node to another, or stay in same node if this is allow by the hypertext system.

• **Navigation**
  
  process of moving through the hypertext.

• **Node**
  
  discrete unit of information. It can be some document, chapter, appendix, card, frame and so on.

• **Reader**
  
  a man who uses hypertext but usually cannot change it. Some systems (e. g. ToolBook) allow readers to make private annotations.
• **Source**
  a place where a link begins.

• **Target**
  a place where link points to.

### 2.3 Advantages of hypertext

Using an hypertext, we have more possibilities to bring information to readers. Hypertext is an ideal way to present information on computers. Hypertext is a powerful tool for authors. It gives many possibilities to information management, education, advertisement, news reporting, electronic searching and so on. Hypertext is less limited as for example books. Authors can add some multimedia features, allows searching in their hypertext systems, and can guide users to other pieces of information very easy. Some of these features are possible only on computers. For example in computer games, we can find houses where the roof is connected with the cellar and vice versa. This is hard to imagine it in real life. So we can create hypertext documents with multi-language support, allow users to change the background to their favorite color, add the ability to turn characters at 90 degree to the left, to read chapters sorted by size and so on. Everything, which can be logically described, is possible. Of course, we probably do not want to do such crazy things like read paragraphs in random order, but some authors can be for example pleased if they can continue in their books with more than one chapter. This is impossible in linear text, but in hypertext, it could be done very easy.

### 2.4 Problems in hypertext

It is useful for us to describe some problems, which we can experience while creating hypertext. If there is no problem, we do not need to look for solutions. As in programs, in small hypertexts we find only small problems, usually syntactic errors. However, if our hypertext system grows up, this becomes dangerous for inexperienced authors. It is similar to creating huge software products. We must realize that writing hypertext is not so simple as writing normal documents. When we write a book, we create chapters and join them together in order. We can create nodes of hypertext as easy as chapters of the
book, but if we try to join them, we can fail, because hypertext is nonlinear and every node can have many links.

2.4.1 Syntactic errors

- **Dangling links**

  Dangling links stay in an hypertext usually after some changes, or in unfinished hypertexts. Good hypertext-systems can detect them automatically, so the author can correct any error at once and easy. However, this shows that making changes to big hypertexts is not so easy. Figure 3 shows an hypertext with dangling link.

![Figure 3. Hypertext with link to deleted node.](image)

- **Disconnected hypertext**

  If we cannot reach all nodes of an hypertext, we speak about an disconnected hypertext. In that case, the hypertext consists of some parts, which are not connected to each other. We can check easily on computers if it has unreachable nodes. On the other hand, in some special cases, we do not want to have circles in our graphs and we want to make from hypertext a tree. This could be also checked very simply. Example of disconnected hypertext problem is in Figure 4.
Introduction to hypertext

Figure 4. Hypertext graph and its two disconnected parts.

2.4.2 Semantic errors

Semantic errors are a much bigger problem and it is not so easy to define them.

- **Lost in hypertext problem**
  
  A main problem is called 'lost in hypertext problem' or 'disorientation problem'. An author creates an hypertext to bring information to readers so that they can access this information in an easy and quick way. Nevertheless, it can happen, that users cannot recognize where they are in hypertext, they cannot find any way to the information they need. They can have problem to decide if the wanted information exists in the hypertext or they do not know which link to choose. They are lost and hopeless. This problem cannot be solved by any algorithm we know.

- **Goto problem**

  This problem arises, when we use links of "goto" type in our hypertext. Today's hypertext systems allow "goto" links and use them often. It is not surprising, that users identify goto facilities with the main cause of their disorientation, and that there is great effort to eliminate "goto" links and prefer for example pop-up notes, expansion links or glossary references. Usage of "goto" links has a parallel in programming language in usage of goto statement, use it only if it is necessary and be careful.
Introduction to hypertext

- **Chunking problem**

  This problem arises, when the hypertext system uses some type of cards with limited length of text. It forces authors to divide their text into parts and this leads to the situation that users can view these parts in different contexts, or can read them in some different ways. This can be inappropriate for the author's intention. Chunking problem demonstration is in **Figure 5**.

![Figure 5](image-url)

**Figure 5.** Division of the one document to the three parts.

- **Framing problem**

  This problem is similar to the chunking problem, but at a more macroscopic level. When writing an hypertext we want to collect information into parts by some topics, link them together to some clusters for easiness of access of related information, and use only a few links to external collections. However, an individual user's interests may be the intersection of some collections. For example, if we are interesting in bio-physics, we cannot be pleased by an hypertext, which is divided into biological and physical parts, when we can jump from one part to the other only over the main menu and we do not have the possibility to follow any other links between these parts. **Figure 6** explains the problem of making collections.
2.5 Quality factors

All efforts in this diploma work leads to making good hypertexts. However, what is it? When we can say, that hypertext system is good? It is necessary to define some factors of hypertext quality, which we can use later to evaluate some hypertext systems. Here is a description of some external factors of program quality [Šešera-93] and than its modification for hypertext systems as mentioned in [Herceg-96].

2.5.1 External factors of programs quality

The external factors of program quality are factors considered by the user. The user is not only the final user, but also every person using the software product. It could be a programmer, who makes a local version of the product or manager of the program and so on. Some of the external factors of program quality are:

- **Correctness**
  
is the ability of the software system to perform the task defined by the user's claim and in the specification.

- **Robustness**
  
is the ability of the software system to work even under abnormal conditions. It doesn't means, that the program will do, what we want, when we give it erroneous data, but it can define some reasonable behavior for this situation.
Introduction to hypertext

- **Flexibility**
  means the easiness of the adaptation of a software system to changes in the specification. There are two ways to raise the flexibility: **simplicity of the system design** (simple architecture) and **decentralization of the system design**.

- **Reusability**
  is the ability of the software system or its part to be used again in some new applications or systems.

- **Compatibility**
  means the easiness to connect the system to other software systems.

- **Efficiency**
  describes how effective (suitable) usage of the hardware components is.

- **Verifiability**
  is the easiness to use of automatic procedures and test data to detect errors in the system.

- **Integrity**
  is the ability of the software system to protect its parts from illegal access and modification.

2.5.2 External factors of the quality from the hypertext view

- **Correctness**
  is a natural requirement for hypertext too. We understand the hypertext correctness as the ability to fulfill defined tasks. We cannot talk about "good" hypertext and do not request correctness.

- **Robustness**
  is also an important factor. However, robustness depends on the program implementation and organization of hypertext systems. Therefore, when an author decides to use some hypertext system, robustness is the factor, that he cannot improve in most cases.
Introduction to hypertext

• **Flexibility**

Today's hypertext systems have an effort to be used everywhere, under every OS and system architecture, independently from hardware and all protocols. They want to have opportunities to run in different modes, for example ability to change some graphics design options if their users wishes so or to display functions of the system dependant on the chosen level of user experience. This means, that flexibility is a very important factor of quality of the hypertext. We must not forget an important set of hypertexts that are changed frequently, for example hypertext news. We cannot use such systems without good support of hypertext versioning.

• **Reusability**

is not so important as in programs, but we can find it useful to link already existing hypertexts to a new one and so to reuse it. We can also reuse only some parts of the hypertext. If we have an hypertext with content, which will not change for a long time, we can use it in other hypertexts. This is reuse of the data part. It is also possible to reuse the presentation structure, if the hypertext system allows use and store of the templates and the other presentation structures. Every hypertext also contains logical structure. This structure can be also reused, especially in hierarchically organized hypertext.

• **Compatibility**

is the next important quality factor. For hypertext systems, this means the easiness to connect one hypertext system to another. This enables some transformations and interchanges.

• **Efficiency**

depends on the program implementation of the hypertext system and it is not a remarkable factor of the hypertext systems. This is true for local hypertext systems. If we look at distributed hypertext systems such as WWW, where data must go through slow links, each author can improve efficiency of his hypertext by choosing proper quantity of data. For example, this can be done by minimal usage of pictures.
• Verifiability

This criterion has the same importance as in programs. It is useful to use special automatic testing procedures in hypertext. Some special structures of hypertext can support this mechanism. One such structure and mechanisms are described in [Banas-96].

• Integrity

is the ability of the hypertext to protect its own parts from illegal access and modification. Today it becomes a very important factor of hypertext systems because of the insufficient degree of security in widely used hypertext systems such as HTML with Microsoft Active X support. Nevertheless, it can be also used for providing information from nodes dependant on the user's degree of understanding, for providing the personal nodes or links etc.
3 Summary of OOP principles

3.1 Conception of object

The first appearance of the notion of an object as a programming construct was in Simula, a language for programming computer simulations. Simula used software objects to represent objects of simulation. The usage of objects for prototyping and application development starts with Smalltalk system, which popularized object-oriented programming.

Object-oriented programming is based on creating and using objects. Objects share their identity and a set of internal states. To emphasize the object independence, objects use message passing for communication. This means, that no one can see or manipulate an object's hidden data. Object can only send a message to another object, and the other object itself selects the method by which it will react to the message. Every object is a manager of its own information and has some functions, which use its internal data. We can say, that these functions define the behavior of this object. So in OOP we do not write functions for solving some parts of problem, nor do we create rules which can be used as a solution, but we try to simulate the problem using objects and give to every object "orders", which can lead to solution of problem.

We can compare this method for example to the building of a house. It is hard to give some algorithm, which describes in detailed and accurate steps, the problem of building a house. Moreover, if we do it, this algorithm will look very strange and difficult. It is more natural to define all experts, which we need, give them their data (plans and materials to use) and define their work. Of course, this is not so easy as we can think at first look. We need for example starting conditions for every expert or system for message passing between the experts. When we create proper objects, simulation can start and user cannot recognize if structural paradigm or object-oriented paradigm was used from behavior of program.

3.2 Terminology of object-oriented programming

To go more into details we define now some basic conceptions of object-oriented programs [Šešera-93], which are important for further work:
Summary of OOP principles

- **Class**
  
  describes a set of objects with the same attributes, behavior and relations with the other objects. It is an abstraction of the objects and contains all characteristics of one class of objects. In addition, a class can also have default and shared attribute values. A default value is fetched from an instance of a class, when an application attempts to retrieve the attribute from object and finds none. A shared attribute value appears in all instances of a class, but all instances reference the same copy typically stored in the class representative object. For example, the class "circle" can have shared variable cnt_circle, which contains the current number of objects defined as circles and some functions for changing this variable as increment_circle. The concept of an object class is perhaps the most basic object-oriented reusability mechanism. Classes are used to create new object's variables called instances.

- **Dynamic binding**
  
  means, that a program links symbolic names and physical addresses at runtime. In OOP, this is used by inclusion polymorphism, because the compiler does not know, which method to use. For example at runtime, we can need to use the same function of the class and subclass too, if we changed the value of some instance in program. Then it is necessary to search through the class hierarchy at run time to find a method of an inherited operation.

- **Inheritance**
  
  is the reusability mechanism to share attributes, functions and relations between classes. It has many forms, depending on what we wish to inherit and when and how the inheritance takes place. The best known is class inheritance, which provides a simple and powerful mechanism for defining new classes that inherit properties from existing classes. With single inheritance, a subclass may inherit instance variables and methods of a single parent class, possibly adding some methods and instance variables of its own.

  Suppose, for example, that we have the class Complex_number and we want to display our instances of this class on a two-dimensional grid. We could define a subclass Graphic_complex_number that inherits from Complex_number and adds

18
a display method. Another type of class inheritance is multiple class inheritance, that is, inheritance of a subclass from multiple parents. When we decide to use a multiple inheritance for our new class Graphic\_complex\_number, new class could inherit from both classes Complex\_number and Graphic\_object. New class inherits the behavior of complex number from the class Complex\_number and the display method from Graphic\_object. It is important to allow overloading of inherited methods. A display method is quite specific to an object, and may have to be re-implemented for a subclass that inherits it.

*Inheritance* makes a relation between two classes. For example, subclass can be viewed as a specialization of its parents. When we create subclass *Humans* inheriting from class *Mammals*, it means that everything what is true for *mammals* also holds for *humans*, but not vice versa. Note that this means that every instance of a subclass is also effectively a member of its parent classes. This points out how specialization is distinct from aggregation. It is not valid to define class *Car* that inherits from *body, frame, wheels*, and similar classes, since a *car* is not a *wheel*. A Graphic\_complex\_number, however, is at once both a Graphic\_object and a Complex\_number. *Inheritance* in this case serves not only as a reusability mechanism, but also as a conceptual structuring mechanism.

Another type of inheritance is partial inheritance, which means, that we inherit only some properties from parents and suppress others. This type of inheritance is used for code sharing for example in C++, but can create a mess of a class hierarchy. These types of inheritance are all of a static form. New classes inherit properties when they are defined rather that at run time. We also know dynamic inheritance. For example part inheritance and scope inheritance. Figure 7 demonstrates how a Graphic\_complex\_number can be created.
Summary of OOP principles

- **Instance**
  
is an object, which was created using an object class. Every new instance of a class has its own set of instance variables, and it shares the operation's methods with other instances of its class.

- **Object**
  
is a basic entity of computation. It preserves its data, which specify its state. It also contains a set of functions called methods, which use its data and define available operations with this object. This is the way to join its state and behavior. Object is represented by a set of allowed operations as a visible interface and by a hidden realizations part, which contains the object's data structures and the implementation of the operations.

- **Polymorphism**
  
is the ability to have more than one form. A polymorph function is one that can be applied uniformly to a variety of objects. For example, the same notation can be used to add two integers, two floating-point numbers or two complex numbers, which we define. It is also possible to allow addition of all their combinations. We can create many addition function and automatically use the one we need. The proper function we can choose in accordance with the types of its arguments. Otherwise we can detect the types of arguments and convert them to the one of them, which is more general. For example, when we need to add some floating-point number and some integer, we can simply convert the integer to floating-point format and than use add function for floating-point numbers. These types of

---

**Figure 7. Inheritance hierarchy of classes.**

- **Number**
  - **Real**
  - **Complex**
- **Graphic object**
  - **Graphic complex number**
Summary of OOP principles

Polymorphism are called the *ad-hoc polymorphism*. It's easy to see that this type of polymorphism can be used only for a finite set of types or classes.

There also exist methods of *universal polymorphism*, which are not limited in this way. One of them is a *parametric polymorphism*, which means, that the functions have special arguments for types of data or classes. The functions use these arguments in the program. A simplified form is called *template*. Templates also have arguments for types, but their values are used by the compiler and not at runtime. A simple example is the type of array. The `array` is a *template* with arguments for array type, which can be "integer" or "char" and when we choose one of them, the compiler will automatically generate a suitable program code.

Another method of universal polymorphism is *inclusion polymorphism*. In this method, the variables can be dynamically bound to instances of different object classes. Inclusion polymorphism is characteristic for object-oriented programming. In OOP, an object from a subclass can occur in a program instead of any object from superclasses.

- **Subclass**

  is the class, which is created from another parent class by the *inheritance mechanism*. Each subclass consists of two parts. These are the *inherited part* and *increment*. Increment contains new variables and methods of the *subclass*, which are specific for this class. The *inherited part* of a subclass is not a copy of the father, but has some individuality. We can overload some functions or attributes. For example, if the father is a note and the son is a footnote, we can change the size of the characters for the son and overload the inherited value. If we use the *inheritance mechanism* on some parent class recursively many times, every class that is created is a subclass of the parent class too.

- **Superclass**

  is a class that is used by *inheritance mechanism* to create some *subclass*. From the view of each *subclass*, its parent class is the *superclass* and every class used in the process of inheritance, which lead to this *subclass* is its *superclass*. 
3.3 Object data management and requirements

To conclude this chapter we need to discuss the requirements needed for object-oriented hypertext systems and look closer at the object data management [Cattell-91]. The required abilities are:

- **Unique object identifiers:**
  The ability to generate automatically identifiers for objects. This is very important, because we need to have the possibility to reference each object in our hypertext system. It is nearly impossible to use some object variables for this purpose because of the big differences among various object classes and because the content of these variables can be changed at run time. So we need special identifiers and every object must have one. Human-meaningful names are useful, but they suffer from dilemma that short and efficient ones tend to be least meaningful to users, and longer ones tend to be inefficient to store. Moreover there are objects that have no human-meaningful identifiers, objects generated automatically and so on.

- **Composite objects:**
  The ability to define objects that contain other objects, such that all the subcomponents and their subcomponents act as a single object. Composite objects can be used to represent aggregation. Aggregation is the grouping of parts into a whole, as car can be aggregation of body, frame, wheels, and so on. Attributes may be aggregated to form objects. Objects may be aggregated to form composite objects. Aggregations may be nested many levels. How are objects that are grouped as composite objects different from any other set of objects grouped or connected by relationships? For example, when a composite object is deleted, the hypertext system can automatically delete component objects and their component objects, if the aggregation is nested. Example of an composite object Document is in Figure 8.
Summary of OOP principles

Figure 8. Object Documentation owns the Chapters.

- **References and integrity:**
  
The ability to reference one object from another and to keep the integrity of these references. Objects use reference attributes to represent relationships between objects. They are like pointers in programming languages or like foreign keys in database systems. However, they have important differences. Reference attributes cannot be corrupted, whereas pointers can be. Referenced values are invalidated automatically when the referenced object is deleted. References attributes are not associated with a user-visible value, whereas foreign keys are. All the values in referenced object may be changed, and the reference attribute will still point to the same object. A little example:

  ```java
  Chapter:
  Title:String;
  Number:Integer;
  Doc:Document; // this is reference attribute, each chapter is
  in the document
  }

  Document:
  Title: String;
  Chapters: List[Chapter]; // List is parameterized type and
  Chapters is a collection of references attributes. It
  means, that each document has chapters.
  }
  
The structure, which is defined by this example is presented in Figure 9.
Summary of OOP principles

![Diagram showing relationships between Document and Chapters]

**Figure 9.** Relationships between Document and Chapters.

Other interesting types of attributes are *derived attributes*. This means, that their values are defined procedurally rather than stored explicitly. They have a procedure to be executed when the value is retrieved or assigned. It can be interesting for hypertext systems to support *active data*. *Active data* are attributes, which execute a procedure when a value is needed, and this procedure will compute the proper value.

The most important relationship for an hypertext system is if some objects are linked or not. However, this is also important to represent an hypertext hierarchy, structures, clusters, and so on. These relationships can be represented using the *references attributes*, which we mentioned. To represent a one-to-one relationship between two objects (object classes), we can add a *reference attribute* to each object. If we want to represent a one-to-many relationship, we can add a *reference attribute* to the first object and a set of *references attributes* to the second one. Object-oriented hypertext system should support *inverse references attributes* to represent relationships, especially links.

**Inverse attribute** is a pair of *reference attributes* pointing to each other. This pair must be kept with each other in sync by the object-oriented system. This means, that if we add another *chapter* to the *document* by setting the *chapter's Doc attribute*, the system must add the *chapter* to the *document's Chapters list*. If a *chapter* was moved to another *document*, then all attributes must likewise be updated.
Summary of OOP principles

How can an hypertext system keep relationship integrity? The system may allow explicit deletion and modification of objects and relationships, especially when in author mode, and may maintain automatically the correctness of relationships as seen through all objects. For example, the relationship integrity would allow the user to retrieve all Chapters for a Document, as well as the Document for a Chapter, and deletion of a Chapter, or modification of a Chapter to be associated with a different Document, would cause the corresponding Document objects to be updated to reflect the change.

- **Object-class hierarchy:**

  The ability to define new object types that have all of the attributes of an existing object type, plus some additional data or behavior. This is very useful in any application with complex structure and hierarchy, and hypertext system is one of them. An object-class hierarchy can be used to represent generalization or specialization. This means that if the hypertext is hierarchically organized, we can use the object-class hierarchy to represent hierarchical organization of the hypertext. This ability can lead the authors to create their hypertext more hierarchical. Nevertheless, generally connected hypertext can be represented by the objects too. However, object-class hierarchy cannot be made, because there is no hierarchy to represent. Generally connected hypertext can be represent as a set of the general objects standing for the nodes and the other set of the objects, which can simulate the links between these nodes.

- **Associated procedures:**

  The ability to work with procedures as well as with data. We need this ability in hypertext system to allow users to make some extensions to the system.

- **Object encapsulation’s:**

  The ability of objects to hide its private data and to use only procedures as interface for inter-object communication.
Summary of OOP principles

- Message passing:

  The way, how to share object's data and not break object's encapsulation. Message passing is the communication's paradigm. It allows any object to communicate with any other object. Objects become active in response to a communication act. In effect, threads of control are determined implicitly by message passing. An object-oriented hypertext system must choose the style of message passing it will use. Each of its objects must have their communication part containing some methods for handling events and message passing. It must have support for event handling, for special events and messages objects, for efficient addressing of each object and so on.
4 Description of hypertext systems and models, looking for object-oriented characteristics

4.1 Introduction to hypertext systems and models

Before describing some interesting hypertext systems and models, it could be good to mention some different ways in which hypertext systems can be realized.

When we plan to create a big hypertext, it is necessary to have a reasonable understanding of the relationships between its constituent objects. When we do not satisfy this condition, our new hypertext system will be an "spaghetti hypertext" in most cases. Normally it is necessary to find a balance between the rigors of a strict, directed tree structure on the one hand, and at the other extreme the combinatorial explosion involved in connecting each object to all other objects. Authors should also consider the possible types of links involved.

In theory, one can distinguish three levels of a hypertext system [Campbell and Goodman-88]:

- Presentation level: user interface
- Hypertext Abstract Machine (HAM) level: nodes and links
- Database level: storage, shared data, and network access

To investigate some hypertext systems and models it is good to known something about the inner architecture of hypertext systems. There are many ways, how to implement links. For example Hanson, in proposing his LinkText publishing system, characterizes links as objects with three attributes: from, to and type [Hanson-88]. Hanson presents links like independent objects at the same level as others hypertext objects. However, in many today's hypertext systems links are not independent. They do not exist as special objects or they cannot have any special attributes. For example, in HTML every link is implemented by two anchors. First anchor marks the source as well as the second anchor identifies the target. This is the only way how to present relationships between nodes.

Most links use to be explicit. This means, that they have to be defined by somebody to connect source node with target node. Some systems also provide implicit links, which
are not defined by authors but follow from various properties of the information. *Implicit link* can be made to some dictionary or thesaurus. A classic example of *implicit links* is the automatic glossary-lookup possible in **Intermedia**. The InterLex server provides a link from any word in any Intermedia document to the definition of that word in the dictionary. It could be impossible to make all these links explicitly. Only when the user requests the definition of a word, the system is looking for the link.

Another type of *implicit links* is **computed links**. An example of the **computed link** is a link from a tourist guide like Glasgow Online to the train schedule, where the system could link to a proposal for the next train out of Glasgow for every destination. Another type of *implicit links* is **back link** in system, which allows only one-way links. Hypertext system can permit to jump to the starting node from any place. This is also an example of *implicit links*.

Another way how to distinguish links is if they are *bi-directional* or *unidirectional*. In addition to standard links connecting two nodes, some hypertext systems also have **super-links** to connect a larger number of nodes. There are several possibilities how to use these links. The two simplest ones are either to show a menu of the links or to go to all destinations at once by opening more windows.

Most current hypertext systems have plain links, which are just connections between two nodes (anchors). Alternatively, a link can be tagged with keywords or semantic attributes such as name of the creator or the date it was created. These tags allow users to reduce hypertext search complexity through filter queries. Links can also be typed to distinguish among different forms of relationships between nodes. We can recognize two basic sorts of links: **tree items** (structural elements) and **annotations** (associative or cross-references). However, many hypertext systems provide several other kinds of link, and may even allow users to define their own, arbitrary link types. Trigg [Trigg-83] presents a very elaborate taxonomy of 75 different link types, including **abstraction**, **example**, **formalization**, **application**, **rewrite**, **simplification**, **refutation**, **support**, and **data**. It is easy to see that providing **typed links** can help user to **navigate** through the hypertext because of additional information hidden in **graphical links**.

Another way in which links may be distinguished is in the level of granularity of the objects that can be retrieved from an access point. For instance, some systems allow links
to reference a specific location or object in document (e.g. Guide, HyperCard, HTML) while others merely reference whole documents or frames (e.g. KMS).

Note that systems, which support typed links, provide a valuable means of locating and grouping materials - every link type is regarded as an attribute, and can be employed by cluster analysis techniques.

Hypertext documents can be broken up into non-sequential units in three fundamentally different ways. In the first, documents retain their linear structure, and can be conventionally read by scrolling and paging (e.g. HTML). However, embedded links allow non-sequential jumps to other sections or other documents. Using this way, we have to "fight" with a Goto Problem very probably. In the second, documents are divided into discrete chunks, determined either by the author or by the limitations of the screen display. This way of realizing a hypertext can lead to the Chunking Problem.

The third and most ambitious form of hypertext is what Ted Nelson calls "compound" [Nelson-81] (realized by Maurer HyperWave, also mentioned in HTML 4.0 specification). Although the conventions of document-level division remain, the emphasis is on allowing users to combine "fragment showers" of information into seamless, virtual documents consisting both of bytes which are native to the central document, and also bytes which are included ad hoc from other external sources. It is also possible to have computed nodes generated for the readers by the system. It could be for example a node with the current weather forecast retrieved from a videotext system like the French Minitel or from WWW page using some CGI script (Author of this diploma work has some experience from developing of such applications).

The hypertext hierarchy or structure is one of its most important characteristics. We can use a strict hierarchy such as trees or lists, where every object has only one link accessing it and is the only object accessing its children but this can be a too big restriction or we can use an hierarchy without restrictions and allow to link every node to every other nodes, but this can be virtually unusable. However it is not a good idea to have no rules for hierarchy of hypertext and keep decision about hierarchy to author free will, because an hierarchical hypertext can be particularly useful at a user's conceptual level by providing a manageble structure in large documents or applications; at
Description of hypertext systems and models,...

progressively higher levels in the hierarchy increasing abstraction occurs, and thus complexity decreasing.

This leads to efforts to propose some compromised hierarchies. In practice, most hypertext authors limit themselves to a maximum of five explicit outlinks from any one node; above this, destinations become difficult to display simultaneously, and the number of links begins to strain user's cognitive capacities. This rule has some exceptions at menu nodes, lists and so on. Note also that as the breadth of the decision tree grows, the action of choosing a side branching link may lead to users failing ever to reach the sub-topic they really wanted or that with choosing some hierarchies for our hypertext system we can fall in the "Framing Problem". Pictures of some possible hypertext structures:

**Figure 10.** Strict (tree) hierarchy.

**Figure 11.** Compromised hierarchy.

**Figure 12.** Overlapping hierarchy.
4.2 Description of hypertext systems

Now we can look at descriptions of some existing hypertext systems and models. There exist many hypertext and hypermedia systems today, so here we describe only a small part of them, which can be used as examples, or which are interesting because of their object-oriented features. Descriptions of hypertext models are from [Herceg-96] and [Banas-96]. They are also mentioned in [Kravčík-95].

4.2.1 Dexter Hypertext Reference Model

The *Dexter Hypertext Reference Model* [Halasz-90] is the most known and widely published hypertext model. It was developed by the designers and representatives of many widely distributed and commonly used hypertext systems on several workshops and conferences. Therefore, it contains some important abstractions from many hypertext systems. This model is formalized in the "Z" language that is the specification language based on set theory.

The *Dexter model* contains three layers:

- **Run-time Layer**
  gives the mechanisms supporting the user's interaction with an hypertext. The presentation of the *components* is in this layer called *instantiation*. A given *component* can have several *instantiations* and each *instantiation* has specified its own number called *instantiation identifier* (IID). The runtime layer contains another entity called *session*, which secures the interconnection between the *component* and its *instantiation*.

- **Storage Layer**
  is a layer describing the "database" containing the network of the links and nodes that is the static structure of a hypertext as a finite set of *components*. The fundamental entity of this middle layer is the *base component*. This *component* can be either:
    - **atomic component**
      is a primitive unit, or a what is usually called "node". Its structure and content is the concern of the *Within Component Layer*. 

---

31
Description of hypertext systems and models,...

- **link**
  
  is an entity representing a relation between two *components*. The link is a sequence of two or more endpoint specifications, each of which refers to a *component* or its part. Every endpoint specification contains the *component specification* or the globally unique identity of it (its UID), the number of an anchor, a direction, and a specification of the presentation.

  The direction determines whether the link points to, from, bi-direct, or none (all these directions are concerning the specified *component*). The presentation specification is some interconnection between the *run-time layer* and the *storage layer* and determines how the link should be presented to users.

- **composite component**
  
  is constructed out of other *components* (atomic, link, composite) creating a hierarchy of *components* - direct-acyclic graph (DAG). This restriction (DAG) means that a *component* can be a *subcomponent* of several (multiple) *components*, but no *composite component* may contain itself either directly or indirectly.

  The connecting of the *components* is solved by using the unique identifier (UID). This is some special system number assigned to each *component*. Both functions working on the *storage layer* work with this unique identifier. The accessor function maps a UID into the *component* being marked with this UID. On the contrary, the resolver function enables some "indirect addressing" by resolving *component* specifications into UIDs.

  Links besides the connections of *components* enable the connection between the locations within the content of a *component*. Anchors are some special mechanism providing this service while separating the *storage* and *within-component layers*. These anchors are some interface between those two layers. An anchor consists of two parts: an anchor ID *a* and an anchor value *b*. The anchor ID *a* uniquely identifies the concrete anchor within the scope of its component and represents the connection of the anchor in the storage layer (the anchor ID together with UID uniquely determines all the anchors in the *storage layer*
Description of hypertext systems and models,...

context). The anchor value $b$ is some arbitrary value exactly determining some position, item, or structure within a component. In contrast to anchor ID, this value represents the connection of the anchor to the Within-component layer.

This mechanism provides the capture of two different paradigms in hypertexts:

- the card paradigm, where the destination of a link is the whole node called card, is represented by a Storage layer’s link pointing to a component. Figure 13 demonstrates card paradigm of hypertext. Links can point only to whole card.

![Figure 13. Card paradigm of hypertext.](image)

- the text paradigm, where a link can point to any position in the destination node, is represented by the composition of a Storage layer’s link and an anchor. Figure 14 demonstrates text paradigm of hypertext. Links can point into nodes too.

![Figure 14. Text paradigm of hypertext.](image)
A component is a complex entity in the Dexter model containing besides the above-described basic component some associated component information. This component information contains:

- a sequence of anchors that index into the component
- a presentation specification containing information for the run-time layer about the component presentation to users
- a set of arbitrary helpful attributes

Besides this data model storage layer defines the set of functions allowing the creation and modification of an hypertext. Here are described the functions provided by Storage Layer in this model:

- **CreateComponent** - creates a new component and adds it to the hypertext. Ensures that the range of the accessor function is extended to include the new component. The resolver function is also extended so that there is at least one specifier for the new component's corresponding UID.
- **CreateAtomicComponent** - takes an atom and a presentation specification and uses CreateComponent to create a new atomic component.
- **CreateLinkComponent** - takes a link and a presentation specification and uses CreateComponent to create a new link component.
- **CreateCompositeComponent** - takes a collection of base components and a presentation specification and uses CreateComponent to create a new composite component.
- **CreateNewComponent** - invoked from the run-time layer, calls one of CreateAtomicComponent, CreateLinkComponent, or CreateCompositeComponent.
- **DeleteComponent** - deletes a component, ensuring that any links whose specifiers resolve to that component are removed.
Description of hypertext systems and models,...

- ModifyComponent - modifies a component, ensuring that its associated information and its type remains unchanged, and that the resulting hypertext remains link consistent.
- GetComponent - takes a UID and uses the accessor function to return a component. If the UID represents a link component, returns either a source or a destination specifier for that component.
- AttributeValue - takes a component UID and an attribute, and returns the value of the attribute.
- SetAttributeValue - takes a component UID, a value and an attribute, and sets the value of the attribute.
- AllAttributes - returns the set of all component attributes.
- LinksToAnchor - takes an anchor and its containing component, and returns the set of links that refer to the anchor.
- LinksTo - takes a hypertext and a component UID, and returns the UIDs of links resolving to that component.

- Within Component Layer
  covers the content and structures within hypertext nodes.

4.2.2 LANGE Formal Model of Hypertext

This model [Lange-90] was firstly presented on January 1990 in the Hypertext Standardization Workshop. It consists of three abstract data types: nodes, networks and structures. And these types are applied to the concepts of object oriented databases.

- nodes
  consist of unique named slots, each having some kind of textual information (a slot is a template for the content of the node) and handles which are parts of the text inside a slot to which a link can be attached.

- links
  have two or more endpoints. They point to nodes, slots, or handles, but also to links or functions interpreted by following the link.
Description of hypertext systems and models,...

- **networks**
  consists of nodes interconnected by relational links.

- **structures**
  describe the organization of nodes and links. The application of the top-down approach to hypertext design leads to defining the structures (containing only the names of nodes and the networks can support the reusability of nodes and networks in several structures).

The *set-oriented structure* represents an unordered collection, *map-structure* adds direct access by user defined names, and the *sequence-structure* expresses interrelationships between nodes. By recursive definitions of these structures, we can create a *tree structure* of nodes.

The LANGE Formal Model does not specify the presentation, it deals only with structural concepts and the other questions and problems are moved to the process of the application design.

**4.2.3 Hypertext Abstract Machine**

One of the first hypertext models was the *Hypertext Abstract Machine* (HAM), firstly presented in 1988 [Campbell and Goodman-88]. HAM is a multi-user server for a hypertext storage system and provides a general model to implement different hypertext applications.

The model consists of five basic objects:

- **node**
  is a basic component containing data. It can be archived, non-archived or append-only depending on status.

- **context**
  is a component capturing the support of configurations, private workspaces and version history trees. Two or more contexts create a tree structure of an hypertext graph. Each context in such graph has one parent (except the root context) and none, one or more child contexts.

- **graph**
  is the highest level object of this model. It contains one or more contexts.
Description of hypertext systems and models,...

- **link**
  
  represents a relationship between two nodes. These two nodes need not to be in the same context, so links can define a more complex internal structure of a graph.

- **attributes**
  
  can specify some properties, values or states of contexts, nodes or links.

Nodes, links and contexts are basic structural elements of the Hypertext Abstract Machine Model.

4.2.4 HM Model

The *HM model* is a hypertext model designed for HM-card hypertext system. This model is described in [Maurer-94] or in [Banas-96]. The *HM model* contains media objects (various types of basic multimedia information), pages (linear lists of media objects, basic blocks of information) and collections (structured objects with unique identifier from which the information database is built). From the structural point of view, several predefined subclasses of a collection are very important and interesting. These subclasses represent several particular topologies of encapsulated links. These four are:

- **envelope**
  
  every component is connected to every other component. All components of an *envelope* are fully related.

  ![Figure 15. Graph representing an envelope.](image)

- **menu**
  
  the head of *menu* is linked to all other members of this *menu* and each member is connected to the head.
Description of hypertext systems and models,...

Figure 16. Graph representing a menu.

- **folder**
  
an ordered set of members, where each member is linked to the previous and next member.

Figure 17. Graph representing a folder.

- **freelinks**
  
a set of members arbitrarily connected by links inserted manually.

Figure 18. Graph representing freelinks structure.

The **HM model** provides a higher level of abstraction than models involving arbitrary links. It is analogous to programming languages, where abstract data types offer a higher level of abstraction than basic data types. Any collection can be a member of any other collection and this recursive membership is meaningful. Collection can be member of
more than only one collection. So larger structures can be used in different parts of the hypertext without multiply storing of these structures.

4.3 Description of hypertext systems

After descriptions of hypertext models, here are descriptions of some existing hypertext systems as they are described for example in [Woodhead-91] and [Nielsen-93].

4.3.1 KMS

KMS or Knowledge Management System is the commercial version of an earlier system, ZOG, developed at Carnegie-Mellon University. It is frame-based rather than node-based, and supports two link types - hierarchical and cross-referential, which are distinguished to aid browsing. Links are one-way, and to whole frames rather than to smaller items. KMS has very simple data structure and the frame is the only existing node type. Links are implemented as properties of frames rather than objects in their own right. Navigation is via backtracking, Goto or Home commands. KMS has rapid text search and indexing, because very rapid access time was a development priority. The development includes fuzzy search criteria, and extended schemas or templates for frame standardization. KMS can also be extended by means of a flexible, high-level development language. Among other features are for example system management or distribution security.

KMS does not provide an overview diagram but instead relies on fast navigation and a hierarchical structure of the nodes. Links across the hierarchy are prefixed with an "@" to let users known that they are moving to another part of the information space. KMS has a special "home" frame, which is directly accessible from any location in hypertext, and allows backtracking to the previous node by single-clicking the mouse as long as it points to an empty space on the screen.

4.3.2 Guide

Guide was originally developed at the University of Kent [Brown-87], but the commercial version is now produced by Office Workstations Limited. Guide is now available for both the IBM PC/AT and the Apple Macintosh. Guide is based on scrolling text windows instead of showing fixed frames. Link anchors are associated with text
strings, buttons or invisible buttons in graphics and move over the screen as the user scrolls or edits the text. This is in contrast with HyperCard, where anchors are fixed graphics regions on the screen.

Buttons are defined as embedded text items, or as graphics items. Text buttons are then denoted by means of default or user-defined typographic styles - bold, italic, underlined, etc. In addition, the mouse cursor changes shape distinctively when it is over one of these active text buttons, or a comparable graphic button. Buttons are created in a two-stage process - by specifying the items at start and end points. An end point is not necessary a new window. It could be some specific location in document. Guide uses the text paradigm. It is also possible to create multiple links to one item, except with an expansion button, which has a unique provenance hidden behind it until activated. Guide support three link types, which are represented by different button types.

- **The replacement button**
  
is used for in-line expansion of the text of the anchor to current hypertext document. It is analogous to nested or hierarchical links. Replacement buttons form a hierarchical structure of text and they can be used to represent for example chapters, sections and subsections. The user can also close an expanded replacement button and then return to the original text. The replacement button exists in a variation called inquiry replacement, which is used to list several options and have the user choose one. This mechanism allows a multiple link type as a super-link.

- **The note button**
  
is a pop-up note that appears for the duration of a mouse-click. This facility is useful for footnote-type annotations, which are closely connected to the information in the main window. Notes cannot contain further anchors.

- **The reference button**
  
is used to jump to another location in the hypertext. To go back to the departure point, users have to click a special backtrack icon (back link). This is analogous to the "goto" link type. It is interesting that the original design of Guide has no "goto" link. It was added in commercial version.
Description of hypertext systems and models,...

Guide has also a simple scripting language, which can be used to launch external programs or to drive peripherals such as video storage devices and modems via an interpreter. To execute these scripts, the special command button must be used. Guide also allows two form of backtracking via icons: back to the start of the original document, or back through a trail of up to 32 recent actions. Text searching is also supported, although only in a forward direction from the user's current position. Every object has its unique numerical identifier.

Guide is controlled via a menu command line, or via shortcuts - using the CTRL button and command initials. Authors can use Guide also for creating their documents, because the Freeze option is available. The Freeze option effectively disables button actions, allowing the author to edit the actual button text or graphic. Authors can also use Show Symbols during debugging, because this option give them more visual cues about identity and type of buttons. We should not forget to mention Guide's Glossary. It is an internal clipboard for items such as buttons and definitions, which authors are likely to reuse. It can be useful for authors to provide a more rapid way of creating hypertext documents.

4.3.3 HyperCard

HyperCard from Apple Computer Inc. is an extremely notable program for its object-oriented features. It uses the standard Macintosh desktop and toolbox metaphors across its applications, which gives it a familiar "look and feel". Applications are structured as cards in "Stacks" (the document, folder or file level), so HyperCard uses card paradigm. A card is the basic node object and a collection of cards is a "stack". Only one card at a time can be displayed. There are several ways of finding your way around a HyperCard stack. Cards are defined in relation to each other and to Home Card. By default, the user can move to the next card created in a linear sequence, or by buttons to more distant cards. The find option will search for text strings - field contents or object names. The recent option provides a card with miniatures of the most recently accessed 42 cards, any of which can be directly accessed. In addition, the Home Card icon is always available to return the user to the top of a stack. There is also the Go menu that can be used at any time to access a specific, known location.
HyperCard defines several distinct user levels, which can be set from the Home Card. In order of increasing power these are:

- **Browsing:** This is HyperCard's read-only level.
- **Typing:** At this level users can add or change text in HyperCard's fields.
- **Painting:** At this level, HyperCard release access to its painting toolbox.
- **Authoring:** This level allows authors to begin to create their own card buttons and fields, and to establish simple card-to-card links. However, the underlying HyperTalk scripts remain transparent.
- **Scripting:** HyperCard's full functionality is released at the scripting level via HyperTalk and all tools from lower user levels.

HyperCard's own application development language is called HyperTalk, which allow users to create their own scripts. In form, it is a high-level, block-structured language, seemingly with features similar to those of Pascal, and to those of the object-oriented language Smalltalk, and indeed to natural language (English). HyperTalk supports message-passing between objects. Objects are stacks, cards, backgrounds, buttons and fields. They have default properties with parameters, which can be manipulated with HyperTalk. Moreover, because HyperTalk is highly tolerant of overloading, and is not strictly typed, objects and scripts tend to be highly re-usable.

The main hypertext support is the ability to construct rectangular buttons on the screen and associate a HyperTalk program with them. This program will often just contain a single line of code written by the user in the form of goto statement to achieve a hypertext jump. Buttons can be activated not only by clicking on them, but also in many others events. For example, when the cursor enters the rectangular region, or when a specified time passed without any user activity. With HyperCard approach to the links, it is easy to create computed links using the HyperTalk language. Anything, what is computable can be used as the destination for a link.

The flexibility of HyperCard can be illustrated on the next example. HyperCard can simulate Guide's pop-up notes by using special show and hide commands in the program definition of the basic links. The designer can determine that a specific text field should normally be hidden from the user but that it will be made visible when the user clicks some button. The result of these manipulations will be very similar to the Guide pop-ups.
In *HyperCard*, an anchor is normally associated with a text string by placing the rectangular region of a button at the same location of the screen as the text string. It is easy to see that this way of placing anchors starts to make troubles, if the author decides to edit text later.

### 4.3.4 Intermedia 3.0

*Intermedia* was developed at Brown University's Institute for Research in Information in Scholarship [Yankelovich-88]. Documents and graphics are displayed in scrolling windows. There are several ways to navigate and browse in *Intermedia*. It has a *Map view* (graphically displayed links), a *Path view* (history of user navigation) and search operations.

*Intermedia* allows the creation of uniform style property sheets and palettes. *Intermedia* links are objects of their own right, with destination properties. An unusual property of *Intermedia* links is that they can have several destinations. Moreover, links are *bi-directional* (there is no difference between source and target anchors), *point-to-point*, and as object properties they are automatically carried over with their owners. Targets can be at any level of granularity from document size to insertion point. Links are indicated by means of icons near to their owners. When a user activates a link from one of its anchors, the system will open a window with the document containing the other anchor and scroll that window until the anchor becomes visible.

### 4.3.5 HTML

*HyperText Markup Language* [Berners-Lee-93] is a set of styles defining various components of hypertext document created in any ASCII text editor. It is derived from the *SGML* (standard generalized markup language) standard. We can look at it both as hypertext model and hypertext system. As model, it has very simple one-way links, which are part of documents as anchors. Any target document cannot see links related to it. As an hypertext system, it offers some interesting features. *HTML* systems use to have only one window. However, it is a text-oriented system, because of the scrolling ability and permission to have link to any point of the document. There exists various *HTML hypertext systems* in many versions, and each one has some special extensions. They allow some multimedia features, frames, channels, styles, object-oriented scripts (JAVA)
Description of hypertext systems and models,...

and many more. HTML grows up and becomes to be more and more object-oriented. As an example can serve CDF, Channel Definition Format added to HTML by Microsoft.

Next lines are from HTML 4.0 specification. We can see that many object-oriented features are added.

- HTML now offers a standard mechanism for embedding generic media objects and applications in HTML documents. The object element (together with its more specific ancestor elements img and applet) provides a mechanism for including images, video, sound, mathematics, specialized applications, and other objects in a document. It also allows authors to specify a hierarchy of alternate renderings for user agents that don't support a specific rendering.

- Style sheets simplify HTML markup and largely relieve HTML of the responsibilities of presentation. They give both authors and users control over the presentation of documents font information, alignment, colors, etc.

- Style information can be specified for individual elements or groups of elements. Style information may be specified in an HTML document or in external style sheets.

- The mechanism for associating a style sheet with a document is independent of the style sheet language.

- Through scripts, authors may create dynamic Web pages (e.g., "smart forms" that react as users fill them out) and use HTML as a means to build networked applications.

- The mechanisms provided to include scripts in an HTML document are independent of the scripting language.

- HTML has its roots in SGML, which has always been a language for the specification of structural markup. As HTML matures more and more of its presentational elements and attributes are being replaced by other mechanisms, in particular style sheets. Experience has shown that separating the structure of a document from its presentational aspects reduces the cost of serving a wide range of platforms, media, etc., and facilitates document revisions.

- Elements may have associated properties, called attributes, which may have values (by default, or set by authors or scripts). Attribute/value pairs appear before the final ">" of an element's start tag. Any number of (legal) attribute/value pairs, separated by spaces, may appear in an element's start tag. They may appear in any order.
In this example, the ID attribute is set for an H1 element:

```html
<H1 id="section1">
This is an identified heading thanks to the id attribute
</H1>
```

- **HTML** has now support for many data types. Here is only a list of the possible link types as a small example:

  - Alternate
  - Stylesheet
  - Start
  - Next
  - Prev
  - Contents
  - Index
  - Glossary
  - Copyright
  - Chapter
  - Section
  - Subsection
  - Appendix
  - Help
  - Bookmark

- Authors may wish to define additional link types not described in this specification. If they do so, they should use a *profile* to cite the conventions used to define the link types.

  - Inheritance of language codes
  - Inheritance of text direction information
Description of hypertext systems and models,...

- The ID attribute may be used to create an anchor at the start tag of any element (including the A element).

This example illustrates the use of the ID attribute to position an anchor in an H2 element. The anchor is linked to via the A element.

You may read more about this in <A href="#section2">Section Two</A>.

...later in the document

<!--BODY-->

...later in the document

<!--BODY-->

<P>Please refer to <A href="#section2">Section Two</A> above for more details.</P>

The ID and NAME attributes share the same name space. This means that they cannot both define an anchor with the same name in the same document.

Reading HTML 4.0 specification we can find many innovations mentioned in the next chapter. Authors of HTML 4.0 try to separate the structure and the presentation of the hypertext. They use many data types, typed links and include possibility to create new link types. The objects can have an ID and these IDs can be used to reference these objects. Cascade inheritance is used to find the proper value for the needed attributes and predefined (default) values are used too. The support of the specific rendering of objects is comparable to the overloading of the Present() method in the next chapter.

4.3.6 ToolBook

ToolBook is an object-oriented application for Microsoft’s Windows. It is designed for users who want to develop their own interactive application. ToolBook uses interactive maps, text and graphic cards, buttons and backtracking. It has its own scripting language, which allows high-level implementation of simple animations, device drivers, and so on. ToolBook is very interesting for us, because it uses objects, object-hierarchies, templates and message passing.

4.3.7 HyperTies

HyperTies is available for both Sun workstations and PC-compatibles. It supports both text and graphics. Anchors are linked with the text. Items are displayed in non-scrolling frames; links are denoted by highlighted text. It has a preview-window, uses a script language and style-sheets.
4.4 Summary of object-oriented characteristics

To complete this chapter, here is a list of object-oriented characteristics found in hypertext models and systems described above ordered by object-oriented characteristics. It is not so easy to decide if some hypertext system has some object-oriented quality. Moreover, there are systems, which fulfil some criteria only partly and some qualities can stay overlooked because of incomplete information. So, accept this list only as author's opinion to this subject.

- **Associated procedures**
  
  They are a very strong feature of HyperCard (HyperTalk) and ToolBook, but we can find them also in Guide, HyperTies and HTML (JAVA, Visual Basic, CGI scripts).

- **Composite objects**
  
  They often occur in hypertext models as Dexter (composite component), Lange (structure type), HAM (graph object contains context objects), HM (HM collections), but it is nearly impossible to find them in hypertext systems. Only HyperCard is so object-oriented that it allows them.

- **Message passing**
  
  Message passing is used in ToolBook and in HyperCard between HyperTalk objects.

- **Object’s encapsulation**
  
  All systems and models have some encapsulation mechanism, but it is hard to say that they support encapsulation in object-oriented way. They encapsulate objects to structures to create some hierarchy and they do not force to use objects in convention of object-oriented approach. And object’s encapsulation is the main feature of any object-oriented approach. Any other object-oriented feature can be skipped over, but not object’s encapsulation. Perhaps the only system that has some support for real object’s encapsulation is Dexter model of hypertext. Its composite object can show the value of its attribute by AttributeValue procedure, but unfortunately, this procedure is not a method of composite object, but a global one.
• **Object-class hierarchy**

The *object-class hierarchy* is best supported by HyperCard's language HyperTalk. Its overloading facility allows it to reuse its defined structures and components. Next are the new versions of HTML with CDF, which has inheritance and class support, and TollBook. HyperTies has style-sheets. We can find some possibilities of recursive use of components in HM's collections and Lange structures type. In addition, the Lange model supports slots, which are a kind of templates.

• **Objects or types support**

Nearly all described hypertext systems know the conception of object. Maybe except the KMS system. However, none of them has full object homogeneity. For example, HyperCard has predefined values in its objects, but it forgot to make objects from links. Links as objects are in Dexter, Lange and HAM models. Moreover, the HAM model has a special object type attribute, which can be used in some other objects. The most interesting from this view is Intermedia 3.0 system, where links are real objects with attributes (to known owners of the link), and are managed by themselves.

• **Unique object identifiers**

Some of hypertext systems partly allow this possibility. For example, Lange model has unique named slot in nodes, Dexter has UID for some objects, and new version of HTML with CDF allow authors to use UIDs, but they are only voluntary and are not generated automatically. In addition, every object in Guide has its unique numerical identifier.

The early hypertext systems were developed before the *object-oriented approach* was popularized. Nevertheless, in today's hypertext systems we can find many *object-oriented features*. It seems that authors of hypertext systems made some efforts to add *object-oriented features* to their systems because of advantages of the *object-oriented approach*, but real *object-oriented hypertext* system cannot be build in this way. HTML extensions are a good example of this. Some improvements can bring *object-oriented feelings* to system, but do not change a paradigm.
5 Consideration of possibilities of using o-o features in development of hypertexts and proposal of class hierarchies for links and nodes in hypertext with o-o features

5.1 Object-oriented hypertext model

Before considering possibilities of using the object-oriented approach in the development of hypertext, it is good to have some object-oriented model to investigate. Let us propose one of this kind. In a previous chapter, there were descriptions of some hypertext systems and models, which have some object-oriented features. However, none of them is completely object-oriented. A natural requirement for the proposed model is to have the all abilities mentioned in the chapter 3.

So, we include into this model:

- Associated procedures
- Composite objects
- Message passing
- Object’s encapsulation
- Object-class hierarchy
- References and integrity
- Unique object identifiers

Every part of the proposed hypertext system must be an object with a unique object identifier. This guarantees that the system will be homogenous. To secure this, there must be one abstract class in the model and every other object must be an instance of some subclass of this class. This abstract class, we can call it Hypertext_object, is the generalization of all used objects in the hypertext model. It means, that it has only attributes and methods, which are common for each hypertext object. These are for example, the unique object identifier, and attributes and methods for message passing. Here is a reduced view of the proposed class Hypertext_object.
Consideration of possibilities of using o-o features in development of hypertexts...

class Hypertext_object:
    uid:Integer;

    method CreateEvent();
    method ManageEvent(e:Event);
}

The language used to define this class is similar to C++ or JAVA. However, it is still only a pseudo-language. It will be used in this chapter to present examples. I was thinking about using JAVA or C++, as predefined and well-known languages. Nevertheless, the great disadvantage of this approach is that it demands the use of too much syntactical sugar. Moreover, these languages have some restrictions, which makes some forms hard to express. Therefore, I decided to use (with some improvements) a pseudo-language, which was founded in [Cattell-91] and readers can already see it in the chapter 3. This pseudo-language is similar to other object-oriented languages and should be self-explaining. Moreover, there will be some commentary after each use of a new construct or statement.

This language is important for the formalization of the object-oriented hypertext model. In a hypertext system following this model, it will lose something of its importance, because of the assumption that every hypertext system of this model will have some object-oriented interface for the developing of hypertext structures. Therefore, there will be no need to use this language so much.

Simply object hierarchy can be defined using the class Hypertext_object. We can see it in Figure 19.

![Figure 19. Hierarchy refined from the class Hypertext_object.](image-url)
Consideration of possibilities of using o-o features in development of hypertexts...

Description of proposed classes:

- **The Data class**
  
is an abstraction of all objects classes, which carry some part of information to present in the hypertext system. From this class are derived classes like Text, Picture, Sound, Music or Video.

- **The Form class**
  
  represents all classes of interface components used in today's hypertext systems. These are all types of buttons and boxes and so on.

- **The Link class**
  
is a superclass for classes of all link types, including implicit link.

- **The Node class**
  
is used to define component objects, which are used to present hypertext structure. They are for example like composite components in Dexter model. At the same time, only one node can be on the screen. Nevertheless, if the hypertext system supports multi-windowing, it is possible to see more nodes, but each one at a different window. A typical instance of the Node class contains Link objects to present cross references, Data objects as its own information displayed when this node is became active, Form objects as additional data, structure and interface support, for example check boxes, list boxes or some kind of buttons, Support objects if necessary and other Nodes objects to stand for organizational links.

- **Support class**
  
  contains every object class, which is not in one of the previous types. It helps to keep the model homogenous. So the Support class is a parent class of Background, Font, Color and many others classes.

Every hypertext created in this hypertext model is an instance of class Node or its subclasses. This instance is the entry value for the browser, which shows its data and forms, and marks all occurrences of links. The browser handles external events like mouse clicks and sends them to the current node. The Node object manages these events on its own right. For example, if the event is a mouse click, the Node can ask its Data and Form instances, if the mouse was clicked over them or not. Node, Link or Support objects cannot be asked in this condition, because they have no display area. Node and Link
Consideration of possibilities of using o-o features in development of hypertexts...

*Objects* contain information about the structure of the hypertext. Where is *Link* instance active is determined by the relationship between the *Link* and some *Data instances*. This means that links use no anchors, but references. They have special attributes, which contains references to linked objects. If any of the *Form instances* is clicked, the *Node* can leave the decision on the next action to it. If a mouse click was on some *Data instance* and this *Data instance* has some link associated with it (*Data object* contains an *reference attribute* to some *Link object*), link should be activated by the *Data instance*.

The Browser can be smart enough and distinguish all object types, but this is not necessary. It can only try to activate all objects it has on its entry. When *Link instance* is activated, it must to know, which object put *Link* to use, and than it could decide in some way, which *Node object* need to be activated next. Than it could simply send this message to the browser. *Message passing* mechanism should be used during whole operation.

Here are pictures of hypertext with normal links (**Figure 20**, anchors must be used to define links) and hypertext, which uses *references* to realize links (**Figure 21**). In the second picture, the top node has *references* to the other nodes; it is a *composite object*:

![Figure 20. Hypertext and its graph.](image)

![Figure 21. The same hypertext with links represented by references.](image)
Consideration of possibilities of using o-o features in development of hypertexts...

To illustrate, how new classes are derived from the parent Hypertext_object class, we can use definition of Node class as an example.

Definition of the Node class for an hypertext system with unidirectional links.

class Node:Hypertext_object{ // Class Node is a subclass of Hypertext_object.
  Links_to_this_node:Set[Link]; // This is a set of incoming links. Each node has references to them.
  Links_from_this_node:Set[Link];
}

Definition of the Node class for hypertext system with bi-directional links.

class Node:Hypertext_object{ // Class Node is subclass of Hypertext_object.
  Owned_links:Set[Link]; // This is a set of links of this node.
}

Instances of the class Node have references only to links, which call this node directly like in the card paradigm of hypertext. Other links can point to nodes indirectly through data objects that they contain. If the hypertext system will use links like that, it will support the text paradigm. This is a similar type of abstraction like in the Dexter model of hypertext. Another abstraction was made by two definitions of the Node class. The aim of this diploma-thesis is not to decide which type of links to use. Moreover, this rule will be applied in the whole chapter. Therefore, there will be sometimes multiple definitions and sometimes only one acceptable solution will be used as an example.

Another class derived from Hypertext_object class is the Data class:

class Data:Hypertext_object{
  Owned_links:Set[Link];

  method Present(); // Abstract method, if the subclass wants to use its own way of presentation, it must overload this method.
}

A naive definition of Text class:

class Text:Data{ // Class text is a subclass of Data class.
  Data: String; // String is not part of our model, used only in the implementation, String should be an object type of the object-oriented language, which the browser supports.
Consideration of possibilities of using o-o features in development of hypertexts...

```java
method Present() {
    printf("%s", Data);
}
}
```

The Text class inherits everything from the Hypertext_object class and the Data class too. Moreover, it overload the Present() method to display contained data on its own will.

A definition of the Link class:

```java
class Link: Hypertext_object { // Class Link is subclass of Hypertext_object.
    Linked_objects: Set[Hypertext_object];
}
```

This definition is too abstract in most cases. Therefore, we need to derive some subclasses of this class, which can be used. However, this class describes some features important for all links. Link knows all objects it refers to (no matter on link type), should manage them itself to keep referential integrity and link can point to every type of object in the hypertext system, also to links, if an author can find this useful.

Some definitions of derived classes:

```java
class One_way_link: Link{
    Source: Hypertext_object;
    Target: Hypertext_object;
}

class Two_way_link: Link{
    Anchor1: Hypertext_object;
    Anchor2: Hypertext_object;
}
```

These new classes still contain the variable Linked_objects and probably never use it. New mechanism to prevent inheritance of unwanted variables and methods can be defined, but it is not so important for proposed model and in many cases, it can be simply overloaded like in the next example. Nevertheless, this works only if the hypertext model has no support for variables with the same names and different types.
Consideration of possibilities of using o-o features in development of hypertexts...

```java
class One_way_link: Link {
    Linked_objects: Ordered_pair;
}
class Two_way_link: Link {
    Linked_objects: Pair;
}
```

### 5.2 Data and schema

We must realize, that every hypertext contains two kinds of information: *data* and *schema*. The *schema* is metadata describing the structure of the *data* and the way of their presentation. It contains information about the structure of the links, the positions of pictures, font types, background color and so on. The *data* themselves are the information, which we want to present and therefore they are most important. Users want to see them; authors want to present them. However, the *schema* is very important too. A good hypertext needs an excellent schema. In today's hypertexts we often find, that *data* and *schema* are created at the same time or that the *schema* is predefined in some way and we cannot change it or make only minor changes. Moreover, this means, that authors do not design their *schemas* at the proper level. They concentrate on creating *data parts* and forget that the real art is to make a *suitable schema* for their *data*. The same hypertext with different *schemas* is presented in **Figure 22** and **Figure 23**. It is easy to see that hypertext in **Figure 23** has more *suitable schema*.

**Figure 22.** Hypertext dictionary with primitive schema.
Consideration of possibilities of using o-o features in development of hypertexts...

**Figure 23.** Hypertext dictionary with more elaborated schema.

In this proposal of an object-oriented hypertext model, a big effort was made to separate the creation of schema and data. The basic rules for the schema are defined in class definitions. The final appearance to the schema authors define, when they create instances of class *Node* or its subclasses. This brings an advantage to hypertext developers, which use to distribute their work to style-designers, graphical-layouters and content-authors. They can work independently. Style-designers and graphical-layouters can propose an object-hierarchy of the classes and implement concrete classes and their presentation methods. Content-authors can prepare raw data and when everything is finished, they could create object's instances and new hypertext is completed.

This process can be illustrated in this simple example:

```java
class Book:Node { // Book is a subclass of class Node.
    Name:Text;
    Author:Text;
    Introduction:Node;
    Chapters:List[Node]; // Chapters are inserted to this list in order, that authors want.
    Contents:Node;
    Index:Node;
}
```

Now we have some information about every possible book. We know some kinds of data, which the books contain and which are important for them. Moreover, we can imagine the basic structure of each book. A typical book has an introduction, chapters, then the contents and an index. It is conceivable to define book with contents at the beginning or make other reorganizations.
Consideration of possibilities of using o-o features in development of hypertexts...

A Book can define its own presentation method and use its data in this method. This can improve graphical layout of every book and be used instead of style-sheets. If we define more subclasses of the class Book with different presentation method, we can use them as graphical templates for books. When the browser is going to display some instance of class Book, chosen instance of Book class can present itself as a first page with name and author of the book, or it can be displayed as a picture of a book with its name and authors written on it.

Now general information about every hypertext book is available. To finish the process of creating a schema, a concrete instance of the Book class must be created, because for example the number of chapters and their divisions are properties of every book itself.

```javascript
object Bible:Book{
    Name = "Bible";
    Author = "God";
    Introduction = Intro_to_bible; // Intro_to_bible can be a instance of the class Introduction.
    Chapters = [Genesis, ..., Apocalypse]; // Imagine, that this is our way of filling a list structure. Genesis can be an instance of class Chapter, but also of class Book. In this list, only subclasses of class Node are required. Therefore, we can use both Chapter and Book instances.
    Contents = Contents_of_bible;
    Index = Index_of_bible;
}
```

Some variables of the object Bible were defined directly; an inline definition was used. To set other variables references, predefined objects were used. The browser shows only one node at the same time, so references to another instances of the Node class or its subclasses can be interpreted as a label of the link between the current node and the referenced node. The browser creates automatically an implicit link, informs this link about its owners and the owners insert the reference to this link to their Owned_links attributes. The schema of object Bible can be changed very easy. Figure 24 contains proposed hierarchy. Figure 25 contains example of an alternative hierarchy.
5.3 Creation of schema in object-oriented hypertext model

When we talk about hypertext, we can see that the most of the drawbacks and problems have something to do with links. Authors have problems to distribute their data into frames, find a proper number of links from one node and so on. Today's hypertext systems use links to create a schema.

Can we find and use some other mechanism to create schemas of hypertexts, especially logical layout of hypertexts? This is analogous to programming and the well-known effort to avoid the use of the goto statement, which also helps in developing of object-oriented programming languages. This leads to development of some methods of programs flow control without goto statement.

Can we use this "No goto" effort and try to create the "No link" hypertext? We must remember that every supposed method must represent the "flow of hypertext". I was thinking about it, but I decided that it could be too much restrictive for hypertext authors. There are some differences between programs and hypertext, which do not allow us to do
Consideration of possibilities of using o-o features in development of hypertexts...

This. If we forbid a programmer to use goto, he can still use other structures and make also good programs, and you cannot recognize from program behavior, if he used goto or not. Nevertheless, if a hypertext author can only use a final set of predefined structures and templates, he probably cannot do everything and hypertext can loose some of its power.

How to solve this problem? We can look once more at programming languages. When people as Niklaus Wirth defined structural programming, they also found some exceptions, where they consider, that goto statement can be useful and they allowed restricted usage of goto statement [Wirth-75]. We can do the same thing in hypertext. If you really feel, that you need a cross-reference link, use it with caution, but using of predefined structures with implicit links is recommended and supported wherever it is possible. In addition, when we give to authors tools for defining their own structures as we did, most of the links can be done in this way. For example, every structural (organizational) link can be represented in a predefined structure of classes. Every kind of menu, dialog, or whatever authors imagine can be created. Nevertheless, they still need explicit links in some cases (annotations). For example to see a picture, which is part of the same node. In addition, an explicit link can be better when some circle of links is created.

You can ask whether this is an important advantage or if we really can implement every structural link in a structure as implicit link. The advantages of using references are described in the chapter 3. These advantages are important enough to explain the effort to replace links with object's references.

Implicit links are created by computer; therefore, authors cannot produce errors in them. Each reference points to some objects, so authors do not need to create special anchors and special bugs in anchors too. Object-oriented systems keep the referential integrity and can invalidate automatically every reference. This means, that an hypertext system can check the hypertext after some changes and prevent authors from breaking integrity of their hypertext (dangling links...). In addition, it is easy to see, that every link, which has meaning as "is a part of", "contains", "includes" can be very easily converted to the reference. For example, a book has chapters; a chapter is a part of a book, so we
Consideration of possibilities of using o-o features in development of hypertexts...

create a proper hypertext structure for a book and a chapter (we give them references to each other).

When we separate some texts to hypertext nodes, we are always able to define a proper structure. It is not the problem to create a subclass of class Node, that represents a "full graph" hypertext, it means every node has a link to every node, and all links can be made automatically and an author has no possibility to make mistakes. When we derive new classes with some restrictions from the Node class, we can define many forms of hypertext. If we have some exceptions (we cannot use only one class type, because our hypertext has some features in different places, which prevent generalization to one class), we can use more types of classes. We can put them together because of their origin. Every one of them is a subclass of the class Node. In most cases, we need to build some hierarchy like in our definition of book. If we need some subchapters, we can use chapters recursively or define a new special class, which can be more suitable.

Little example:

class My_structure:Node{
    Numbers:List[Number];
}

class Number:Node{
    Number:Integer;
    Parent:My_structure; // Inverse attribute to attribute Numbers in My_structure.
    Links:List[Number];

    method Number|){ // This is an constructor. It is called before first use.
        Links += Bigger_or_smaller|Parent.Numbers; // Suppose that this statement will act as described below:
        "Add to Links first Number from Parent.Numbers, which has bigger Number than me. If I am the biggest Number, add smallest Number, but check if I am not only Number in Parent.Numbers. In that case do nothing."
    }
}
Consideration of possibilities of using o-o features in development of hypertexts...

Let's have these instances:

```java
object Number1: Number{
    Number = 1;
    Parent = My_hypertext;
}

...

object Number4: Number{
    Number = 4;
    Parent = My_hypertext;
}

object My_hypertext: My_structure{
    Numbers = [Number1, Number2, Number3, Number4]; // 4 numbers added to list.
}
```

![Diagram of My_hypertext structure]

**Figure 26.** Structure of *My_hypertext*.

The final structure of new hypertext can be seen in Figure 26. From *My_hypertext node*, we can go to every node and vice versa. In addition, every other node is linked to the next and previous node (if we suppose bi-directional links). Moreover, inverse attributes can be easily detected by the system, if bi-directional links are used, because the system will try to create two links with identical content between each *Number instance* and *My_hypertext object*. It is caused by references in these objects to each other. An hypertext system should recognize this and look at these references like to the
Consideration of possibilities of using o-o features in development of hypertexts...

relations between inverse attributes. This means, that when the author moves Number4 from My_hypertext, the reference to My_hypertext in Number4 should be moved too.

Now imagine that we want to change the structure of our hypertext very quickly and do not add many bugs to this hypertext. We can create new subclasses from existing classes and use inheritance or if we do not need the previous schema, simply rewrite the definition of class Number, like in next example:

class Number:Node{
    Number:Integer;
    Parent:My_structure;
    Links:List[Number];

    method Number|() { // This is an constructor. It is called before first use.
        Links += All_bigger_only|Parent.Neighbors; // Suppose that this statement will act as described below:
        "Add to Links all Numbers from Parent.Neighbors, which have bigger Number than me."
    }
}

Figure 27. Changed structure of My_hypertext.

The Figure 27 shows us how much the structure is changed. Moreover, look at allowed links from any Number instances. For example, the browser can show to users links from Number2, which are in attribute Links in the way as we can see it in Figure 28.
Consideration of possibilities of using o-o features in development of hypertexts...

Node number 2
You can choose links to:
Number3
Number4
Back

Figure 28. Displayed text, while browser presents node Number2.

If we rewrite Number Present() method, the result can be very interesting. We can see it in figures 29, 30, 31 and 32.

<table>
<thead>
<tr>
<th>Node number 1</th>
<th>Node number 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &lt; 2</td>
<td>2 &lt; 3</td>
</tr>
<tr>
<td>1 &lt; 3</td>
<td>2 &lt; 4</td>
</tr>
<tr>
<td>1 &lt; 4</td>
<td>Back</td>
</tr>
</tbody>
</table>

| Figure 29              | Figure 30              |

<table>
<thead>
<tr>
<th>Node number 3</th>
<th>Node number 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 &lt; 4</td>
<td>Back</td>
</tr>
</tbody>
</table>

| Figure 31              | Figure 32              |

The new defined schema represents the relation "<" between instances of Number.

We can see that this way of hypertext development gives us the chance to create a hypertext schema from a predefined one and our new classes can be put together in any way we want. In this way, we can build an hypertext more quickly, with less errors and the final result can look better because of using standard parts many times. When we decide to change the schema, it should have no influence on data and vice versa, we can reuse an existing schema for other data.

When we define new classes, this takes some time, but than we can use them many times, save our time and avoid unnecessary bugs. When creating new structures, we can use inheritance and develop new structures more quickly. Some of today's hypertext
Consideration of possibilities of using o-o features in development of hypertexts...

systems allow the use of some type of inheritance (templates), but in most cases only in
graphic parts of hypertext. This is natural, because in graphical applications we can see
the power of inheritance very fast. However, using inheritance for creating schemas can
also be very useful. Why to begin every time from start, when we can use existing
hierarchies? We can just copy old structures, but the mechanism of inheritance allows us
to change them to fit in the best way to our hypertext, which is a big advantage.

5.4 Encapsulation - the main characteristic of the object-oriented
approach

The reader can see that every hypertext proposed in our model is only one object of
the class Node from the "top view". For example, we can have the book Bible. It is an
object of class Book, and class Book is subclass of class Node. When the browser
accesses the object Bible, it can see the data of this object and try to present them on the
screen. For every object of class Node or its subclasses, which the Bible contains, the
browser must create some links, but the browser does not see to these objects at this time.
When some link is activated, the browser will access the object it points to and
everything will repeat. We can say, that the objects we defined are encapsulated. How we
can use this, or can this feature brings us some advantages?

Imagine the case, that we create some more books as in the next example. We use
different hierarchies and styles, other rules etc. However, from the top view these
hypertexts are still some objects of subclasses of class Node. Therefore, when we decide
to make our hypertext library, we need only to define a new structure for the library, as
we like. For example:

class Shelf_for_books:Shelf{ // This new class is a subclass of class
    Shelf. Let us assume, that this class includes some graphics and
    others important information for all shelves and a list of items on
    the shelf, which we overload.
    Items:Alphanumerically_ordered_list[Book];
    Number_of_titles():Integer; // Number_of_titles is a active data
    type. It uses method of the same name, when needed.
    Name:String;
Consideration of possibilities of using o-o features in development of hypertexts...

```java
method Number_of_titles(){ inline definition of needed method
    Number_of_titles = count|Items); // this method work with
    variable Number_of_titles, which is part of the same object and
    with the predefined procedure count for class List or its
    subclasses.
}

object My_first_library:Shelf_for_books{
    Items = [Bible, Koran, Greek_m Mythology] // we put these three books
    to our library. They are instances of class Book.
    name = "My little theological library";
}
```

When we give this object to the browser, it could show us three links in some
g graphical way, but alphanumerically ordered. In addition, we can see the name of our
library and the number of titles, which our library now contains. The computation is done
by a new object type Integer, which is a subclass of class Data and is set as active data.
Class Integer has its own method Present() to show data on the screen. Because variable
Number_of_titles is an active data, the browser knows, that we do not want to use simply
the object Number_of_titles of class Integer, but we want to use the method of the same
name, whenever the browser accesses the object Number_of_titles. This method must set
this object of class Integer, whenever it is called and than the browser can work with
simple Integer object Number_of_titles. The reader can imagine other ways to implement
a feature like this. Another technique how to implement active data is based on use of
inheritance and defining new integer type, which can be used in the class
Shelf_for_books.

```java
class Special_integer:Integer{
    Data:Integer;
    Owner:Shelf;

    method Special_integer(){ // this is an constructor
        Data = count|Shelf.Items); // initializing, it is computed only
        once
    }
}
```

65
Consideration of possibilities of using o-o features in development of hypertexts...

```java
method Present {{{
    Integer, Present |)); / use same method as class Integer
}}
```

We can see that this solution is not so good. We need to define a new special class, which is probably useless for other applications, we must send also the whole list of Items to this object and the computation is running only once and not at run time, which gives us more advantages. For example, if we allow the user to change the content of our shelf during run time, the browser will continue to display the proper value for the variable Number_of_titles.

In our model of hypertext, we can use both top-down and down-top design methods to develop a new hypertext. As we have seen in the example, we can define the books first and then create a library, but if we prefer the top-down design, we can propose some type of library and then give books to it. We can change some parts of the hypertext, add new parts or join some different parts with a new super object without problem. This is an advantage of encapsulation.

Encapsulated objects can act as composite objects. So when we decide to remove, delete, update or insert some parts of hypertext, we can do it very quickly changing only one object and let the object to apply the changes on its parts itself. We can imagine many operations, where this feature can help us. For example, when we resize a button, text on this button will be resized automatically. Moreover, when we delete a node, all links pointing to this node can be deleted too.

5.5 Links as independent objects

Until now, we talked about advantages of object-oriented models of nodes, but which advantages bring to us object-oriented links? In the new object-oriented model, we try to eliminate most of them by implicit links and it seems that they are not so important for us. Nevertheless, we also defined the class Link and now it is the right time to look how to make instances of it.

It is nearly impossible to create a large hypertext and to use only implicit links. These links are used for creating the schema, some structures and hierarchies, but there are still circumstances when we need explicit links, links made by hand. These are for example

Consideration of possibilities of using o-o features in development of hypertexts...

links for guided tours, some types of shortcuts, references to other sources, associations, annotations and many more. Moreover, the class Link gives the authors a tool to create their own link types, which can change the hypertexts as they wish. Some interesting link types, which can be useful, are:

```java
class Super_link:Link{
    Anchor:Abstract_object;
}

class One_way_link_with_additional_information:One_way_link{
    Author:Text;
    Date_of_creation:Time;
}
```

The class Super_link has a variable Anchor for the source part of the link and an inherited variable Linked_objects for the allowed connection. A browser can use any method, which was described before to use a superlink, or the superlink can define its own new method for accessing target objects. The second defined class simply adds some new variables to the One_way_link class and it could bring some new advantages to users, if they try to filter links. Now look at some uncomplicated instance of link.

```java
object My_link:One_way_link{
    Source = My_homepage.Text_about_my_best_friend_homepage;
    Target = My_best_friend_homepage;
}
```

This is a small example how to create explicit link. Source and Target need to be instances of subclasses of the Hypertext_object class. In this example, Source is set to a object of class Text and Target is set to an instance of the Node class. This is a very important feature of our model. We can link everything with everything. We do not create special tags for links which and only which can be used for links. We can have special classes as buttons, which are used for links. In addition, some parts of text can be used as anchors and they do not need to be labeled in some special way.

This method of creating anchors helps us, when we make changes to the hypertext. We can insert new parts of text before talking about My_best_friend_homepage, but the link still starts on this text. When we remove this text, the browser can easily detect a link without start. When we use an object of Node class as Target, the browser starts to
Consideration of possibilities of using o-o features in development of hypertexts...

present this node. However, we can also use some others types of classes and it depends on the browser, what you will see. If the targeted class is text, the browser can show this text for the time that you hold your mouse down. Alternatively, it could play some music, talk to us or show some video. The browser can also use the Present() method of the object and then we can do nearly everything, what we want to be done after clicking on some part of screen. For example, when authors want to create special help for some words in text, which will scroll in the first line of the screen, they could simply create a new Data subclass with a method of presentation which they require for their help and than link the chosen words to objects of this type. This possibility to add special link's targets to the Data objects is comparable with the active texts used in the Oberon-system of N. Wirth.

When the browser finds in some instance a reference to another instance of the Node subclass, it knows that an implicit link is wanted. The browser can simply automatically generate one explicit link object and use it. Therefore, the browser in our model always knows all explicit links and all links from visited nodes (because the browser creates explicit links for them). Links are objects and they need to be managed in some way by the browser. Let us suppose that they are all stored in a simple list of links. This allows us to check them very quickly. We can control the integrity of our hypertext, make some statistics and use them to keep given limitations. For example, if we want to avoid too many links from one node, we can define some maximum number for them and the browser can check it during writing our hypertext. It is possible to create some maps or preview pictures of our hypertext very quickly. We can also filter them. We can show only structural links, which was made implicitly or only links from some authors, of some date or links of some types. It could be interesting for some users and they can change the hypertext, as they want. We can imagine some hypertext encyclopedia about physics and biology where we use special link type:

class Special_link:Link with additional_information{
   Is_to_biology_topic:Boolean;
   Is_to_physics_topic:Boolean;
}

When the user wants to get some information only from the physics part, he can filter the links he is interested in. This can partially solve the Framing problem. Are you
Consideration of possibilities of using o-o features in development of hypertexts...

interested in bio-physic? So use links, which point to nodes with biological and physical
content too. This could be very useful in large hypertext systems. We can implement for
every order: "Never show me links to the nodes I already visited" without any
problem. As you can see, object-oriented links bring us flexibility. We can change the
way in which the browser presents information. What about \textit{computed links}, which are
support by some of the hypertext systems? Look at the example:

```java
class Computed_link:One_way_link{
    Target:|):Node;

    method Target{|
        Target = Find_first_train_to_Bratislava|Current_time);
    }
}
```

The variable Target is of \textit{active data} type. Now when the browser tries to access
Target to use a link, Target will be created and set to the result of the function
\texttt{Find_first_train_to_Bratislava(Time T)}. We can use this link only in a node with
information about trains to Bratislava and therefore it is not very useful, but we can create
a more general link with function \texttt{FindFirstTrain(Time T, String City)}.

\textbf{5.6 Utilization of the other object-oriented features}

Some of the object-oriented programming languages allow programmers to use
\textit{predefined values} for their variables. Let us see how this feature can improve the
development of hypertexts. Some hypertext systems support this innovation (HTML 4.0).
When authors create some instances of an object class, sometimes they are forced to
initialize some variables repeatedly to the same, commonly used values. This is irritating
and wastes author's time. We can create our classes with \textit{predefined values} for some less
important variables, so the authors do not need to care about them, if they do not want to.
For example:

```java
class Text_for_advenced_users:Text{ // All that is important for
text, we inherit from class Text, we add only new features, which
can improve the final presentation of texts.
    Color_of_text:Color = BLACK; // Let Color be class with suitable
    structure for color data.
}
```
Consideration of possibilities of using o-o features in development of hypertexts...

```java
Used_font:Font = "Helvetica";
Bold:Boolean = false;
Italic:Boolean = false;
Underlined:Boolean = false;
Vertical:Boolean = false;
Motion_effect:Boolean = false;

Fog_effect:Boolean = false;
Make_unreadable:Boolean = false;
}
```

We can invent many extensions without making the use of new classes too difficult. We can use the new class `Text_for_advanced_users` as class `Text`, and only if we really want to, we can use some new features. Using predefined values can be valuable, when we create a big hypertext and many people are working on it. Alternatively, we want that all our hypertexts have some equal characteristics. In that case we can create some hypertext classes with predefined values and decide to not overload their values. Than all articles in our hypertext magazine have the same font used in the title, the same font size of text and we do not need to care about who wrote this article and how to adapt his article for our magazine.

Many object-oriented programming languages support shared variables of classes. With this feature, hypertext system can easily compute number of instances of some class. Some special objects can refuse to create themselves, if they already exist, or if we create some shareable hypertext database, we can know actual number of users or the very popular actual number of access to our page.
6 Summary

6.1 Main results

This diploma work tries to answer two main questions:

1) "What is object-oriented hypertext?" or when we can talk about some hypertext system as about an object-oriented hypertext system? One object-oriented hypertext model was proposed to present the author's opinion to this topic. Some requirements for object-oriented hypertext systems were defined.

2) "How can object-oriented approach helps in the development of hypertexts?" or can it be useful for authors and readers? This work tries to persuade readers that object-oriented hypertexts are not only another form of hypertexts, but that this is a next step in the hypertext development. Some existing hypertexts systems were checked for object-oriented characteristics and this examination was partly successful. Many object-oriented characteristics were discussed and they were found valuable for hypertext systems. This can be viewed at used examples. Moreover, the evolution of existing hypertext systems (HTML) proves the advantage of object-oriented approach too.

6.2 Quality factors of object-oriented hypertext systems

So, we proposed an object-oriented hypertext model, discussed some problems, which can appear in creating a concrete object-oriented hypertext system and consider the usability of some object-oriented features like inheritance, encapsulation, shared variables of classes and so on. To bring to an end this investigation of possibilities of using object-oriented approach in developing of hypertext, let us discuss external factors of hypertext quality, from the view of the proposed object-oriented hypertext model:

- **Correctness**

  The correctness of an object-oriented hypertext system depends on the good implementation of all required abilities (for example message passing) and on authors. Like in programming, the object-oriented approach cannot guarantee the correctness of program, either object-oriented hypertext cannot promise correctness of written hypertext. If an author prefers the object-oriented approach, an o-o hypertext system can help him to keep his hypertext correct.
• **Robustness**

In today's hypertext systems, the robustness cannot be improved by the authors. However, the possibility of overloading many primary methods in the proposed model can allow the author to increase robustness of his hypertext. He can change the constructors of all objects or the *Present() method*, and handling situations, that was forgotten in the hypertext system.

• **Flexibility**

The proposed model has the ability to change nearly everything. The author can overload many methods or derive his own subclasses from existing classes and this feature can increase the flexibility of the hypertext system. Let us suppose that the specification of a developed hypertext has changed. For example, an author must increase security level of his hypertext. If he need to add password protection to some part of the hypertext, he could create a new special link class, which will have a constructor method to check the password before it will permit to browser to access the next part of the hypertext.

• **Reusability**

The object-oriented approach promises very high reusability. Really, a data object can be used as data in many other hypertexts without any changes and if the author likes the schema he created, he can use it once again with other data or very simply inherit a more suitable schema for new data from existing schemas.

• **Compatibility**

Compatibility is a quality of concrete implementation. We cannot talk about it at the hypertext model level; we need some existing hypertext system. However, it is easy to see that implementations of the proposed hypertext model should be able to simulate any other existing hypertext system because of their big flexibility.

• **Efficiency**

Like compatibility, efficiency can be checked only for a concrete application. We can suppose that the efficiency of object-oriented hypertext system will be lower because of the *encapsulation, message passing* and so on. Nevertheless, this is not so significant because of the good performance of today's computers.
Summary

- **Verifiability**

  When programmer use C++, this does not mean, that the program will be easy to read, understand or check. This is valid for hypertext systems too. However, the distribution of hypertext development into two parts (schema and data creating) and the object-oriented approach gives authors tools to keep their hypertext "controllable".

- **Integrity**

  Integrity can be a big problem of object-oriented hypertext systems because of their ability to create new methods. If only trusted authors have the privilege to change hypertext, everything is all right. Nevertheless, if the hypertext system allows any user to put its own methods to the hypertext, this means to make many protection precautions in the used scripting language. And this could be very difficult.
7 Literature

7.1 Literature in paper form

- [Herceg-96] P.Herceg, "Differences between the structure of programs and hypertexts", diploma-thesis, Faculty of Mathematics and Physics, Comenius University, Bratislava 1996.
7.2 Electronic documents

- HTML 4.0 specification, http://www.w3.org/TR/REC-html40
8 Appendix

The diskette with the hypertext version of this diploma work in HTML format.