**Západočeská univerzita v Plzni**

**Fakulta filozofická**

# Bakalářská práce

**2014** **Monika Chladová**

**Západočeská univerzita v Plzni**

**Fakulta filozofická**

**Bakalářská práce**

# TRANSLATION FROM THE FIELD OF INFORMATION TECHNOLOGY AND ANGLICISMS RELATED TO THIS FIELD

**Monika Chladová**

Plzeň 2014

**Západočeská univerzita v Plzni**

**Fakulta filozofická**

Katedra anglického jazyka a literatury

**Studijní program Filologie**

**Studijní obor Cizí jazyky pro komerční praxi**

**Kombinace angličtina – němčina**

**Bakalářská práce**

# TRANSLATION FROM THE FIELD OF INFORMATION TECHNOLOGY AND ANGLICISMS RELATED TO THIS FIELD

**Monika Chladová**

*Vedoucí práce:*

PhDr. Eva Raisová

Katedra anglického jazyka a literatury

Fakulta filozofická Západočeské univerzity v Plzni

Plzeň 2014

Prohlašuji, že jsem práci zpracoval(a) samostatně a použil(a) jen uvedených pramenů a literatury.

*Plzeň, duben 2014* ……

*Ráda bych touto cestou poděkovala vedoucí práce PhDr. Evě Raisové za cenné rady, připomínky a čas, který mi věnovala. Dále bych chtěla poděkovat svým blízkým a rodině za podporu během studia.*

## Table of contents

# 1 Introduction

The objective of this thesis is the translation of the text from the field of IT with a commentary and glossary. The thesis is divided into two main parts - theoretical and practical part. The theoretical part is the processing of basic knowledge of the theory of translation. The theory is based mainly on two books of two Czech significant translatologists and it is "Umění překladu" by Jiří Levý and "Překlad a překládání" by Dagmar Knittlová. The theory of translation deals with the process of translation, equivalence and describes the main types and procedures of translation.

In the next chapter is presented the description of one of functional styles. It is a scientific style, in which the selected text is written. Texts from this field of information technology are predominantly written in scientific style. There are described the main features of the English scientific style according to Dagmar Knittlová. For comparison is mentioned popular scientific style.

All this theoretical knowledge is further applied in the practical part, in which the translation from English into Czech is introduced. For the translation the first chapter of a book was selected. The title is "Object-oriented design with UML and Java" by K. Barclay and J. Savage. This is a book designed especially for programmers. It is expected that the biggest problem will be when translating of technical terms from the field of IT, which have mostly in the Czech language no equivalent. Therefore it will be necessary to suggest for these words suitable Czech translation. In many cases, the words with zero equivalence remain untranslated in the Czech language. These are therefore loanwords and the process how the anglicisms are adopted in the Czech IT field, which accelerates these days due to internationalization. Other particular problems that occurred during translating would be commented on in the microanalysis.

Before the commentary containing microanalysis, where the target text is commented on, the macroanalysis is carried out, which focuses on the analysis of the source text. There is described a source, topic, author, reader, function, structure and style. Then the text is analyzed from the grammatical and lexical point of view. On the basis of this knowledge the functional style is determined. In this part is possible to use knowledge from the theoretical part about the scientific style.

Further a glossary of technical terms is presented that are contained in the selected text. Terms are introduced with possible Czech equivalents and their meaning.

At the end of the thesis is a chapter that devotes to the topic of anglicisms. We can find there a brief characterization of anglicisms, their development and formal adaptation during their adopting into the Czech language. Also the examples related to this field are mentioned there.

# THEORETICAL PART

## 2 Theory of translation

The term translation represents the transfer of information from a source language into the target language. It should take into consideration the grammatical structure of the target language and compliance of style of the text. This is mainly a representation of the content of the original text with adequate resources of target language.

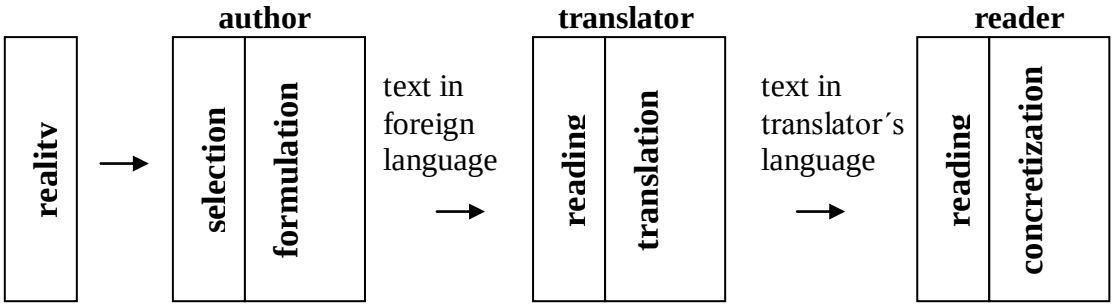
According to J. Levý, the translator should know:

1) the language from which he translates,
2) the language into which he translates,
3) factual content of the translated text (e.g. appropriate field of the scientific literature).

### 2.1 Process of translating

"Translation is communication. More precisely, the translator decodes the communication contained in the text of original author and rewords (encodes) it to his language. Then, the communication of the translated text is deciphered by the reader." [1] This process can be illustrated as follows:

**Picture no. 1:** Process of translating [2]




Earlier translations were evaluated as products, but nowadays the modern approaches focus more on the process. New Anglo-Saxon theories of translation are divided into two approaches to translation. This is a macro approach and micro approach. The macro approach focuses on the relationship of the author to the topic and audience, type and function of text, cultural, historical and local background.

After a strategic decision making comes detailed decision making and then micro approach that analyzes specific examples, grammatical and lexical structure. This leads to the construction of the final target text.

The process of translating is not over, when the text of translation was created. Also the translation works in society only when it is read. For this reason, the translator must take into account the reader, for whom he translates. It comes to the subjective transformation of objective material: firstly, at the author's conception of reality, secondly, at the translator's conception of the original and thirdly, at reader's conception of translation. In other words we can say, that the main point of translation is the relation of three parts: the objective content of the work and its double concretization, the reader of the original and the reader of the translation. These three structures will be necessarily a little different, especially in terms of differences between two languages and differences in contents of consciousness of two readers. The limitations of these differences is the biggest problem of the translator. During the translation are mainly these relationships:

a) **between the source language and target language** - the results of comparative linguistics,
b) **between content and form in original and translation** - methods of literary science and comparative stylistics,
c) **between the final value of the original work and translation** - methods of literary criticism.

## 2.2 Equivalence

Dagmar Knittlová introduces in her book the biggest problem in translating, which is the question of reproductive accuracy, the question of equivalence, which deals with the possibility of transformation of all information from the source language into the target language in spite of differences in the grammatical system of these languages. The main principle of modern translation is the principle of functional equivalence. This means that it is not important what kind of language resources are used, but it is essential that they fulfill the same function. This principle is an optimal relationship between the translation and the original text.

During the translation of scientific literature the emphasis is put on the information content. Language resources have only a communicative function in contrast to art literature, where the aesthetic effect is required. The problem occurs at the incommensurability of language materials of source and target language. It is therefore possible that does not exist complete agreement of meaning of the original and translation and it does not suffice only linguistically correct translation, but the interpretation is necessary to find the most suitable equivalent. This means, for example, specifying the term or its extension.

"The equivalence is ensured with equality of semantic relation to reality. Some element may not be explicitly expressed, equivalent may lose its special attribute and replaces it with another, or take a different character, e.g. expressive or intellectual." [3] During the finding equivalents the translator must focus on work not only with lexical equivalents but also with syntagmatically and enunciation, because some word in one language may not correspond exactly one word in the other language. Therefore, instead of the term equivalent is used rather expression a translating counterpart in the target language.

### *2.2.1 Types of equivalent (according to D. Knittlová)*

1) **full equivalent** - semantic content and extent of the word is the same in both languages. These are names with almost unambiguous denotative meaning indicating basically the same or corresponding part of extralinguistic reality, e.g. eye:oko, window:okno. "It is to be expected that the Slavic verb generally contains more information than the English verb, is semantically richer. It is again related to the typological difference between languages with nominal character of English and rather verbal type of Czech. Translators should not forget and should benefit from the richness which the Czech language provides them in this sense." [4],
2) **partial equivalent** - at least one part of the content and extent of the word is equivalent. English as an isolating analytic language has more analytical, compound and explicit terms than synthetic, flexional Czech language,
3) **zero equivalent** - equivalent does not exist. In this case, it is necessary to replace the empty position by paraphrase, calque or loanword (e.g. midterms:čtvrtletí, computer:počítač/computer, mop:mop), or it is replaced equivalent of situation,

if the society does not know the situation of the source language (e.g. porch:veranda),

4) **more equivalents** - for example go:jít/jet/letět/plout.

## 2.3 Types of translation

According to Dagmar Knittlová, the recipient of the final text perceives the final product as a result of the decision-making process. If it is a good translation, the recipient does not perceive a number of decisions and dilemmas. "A good translation should not be seen as a translation but as an original work created in the given language." [5] The translation should therefore meet the following three criteria:

a) the final translation is absolutely natural,
b) the final translation has the same meaning as the original text,
c) the final translation preserves the same dynamics, causes the same reaction.

The target text is equivalent to the original in terms of meaning, style and hypersyntax. R. Jakobson distinguishes the following three types of translation:

- **intralingual translation** - internal explanations in the text, repeating already said (paraphrase),
- **inter-semiotic translation** - interpretation of information by one semiotic system with another semiotic system sys (reading of ordinary wristwatch),
- **interlingual translation** - interpretation of information of the source language with target language.

Then we can divide the translation into form-based and meaning-based translations. The following types of translation belongs to interlingual translation, from that the first two are form-based and the other two are meaning-based. The examples are used according to Dagmar Knittlová:

- **interlineal translation** - an extreme example of translation that does not respect the grammatical system of the target language (e.g. I did not want to hurt you - Já nechtěl ublížit ty/tobě),
- **literal translation** - sometimes referred to as "slavish", respects the grammatical system, but does not take into account the lexical units such as collocation,

idioms (e.g. I ordered him to brush his teeth - Poručil jsem mu, aby si vykartáčoval zuby), target text might sound strange,

- **free translation** - it is the opposite of the interlineal translation, it respects the source text only partially, it does not take into account the register or stylistics and the content is transferred freely, which causes the change in the meaning.
- **communicative translation** - using of natural formal resources of the target language, especially in respect to translation of greetings and wishes, signs (e.g. Czech good day does not have an equivalent in English, it can match good morning or good afternoon).

"In fact, the resulting translations are mixture of all these types of translation, because it is not easy at all to translate idiomatically consistently." [6]

## 2.4 Translation procedures

Dagmar Knittlová mentions in her book the lack of direct equivalent in the target language, which is solved by these procedures (according to Vinay and Darbelnet):

1) **transcription** - it is used for overwriting of the pronunciation of foreign words; transliteration is the overwriting by different alphabet and it happens to sound distortion (e.g. Maotsetung),
2) **calque** - literal translation into the target language when preserving the structure of the source language (e.g. skyscraper - mrakodrap),
3) **substitution** - replacement of one language resource by another one (e.g. substitution of nouns by personal pronoun),
4) **transposition** - necessary grammatical changes in the language system,
5) **modulation** - shift in the semantic field (e.g. the wonders of Czech cuisine - chuť české kuchyně),
6) **equivalence** - not very properly chosen term for the use of stylistic and structural resources (e.g. a little girl - holčička),
7) **adaptation** - replacement of situation that does not exist in the target language, for example wordplay or proverb,
8) **borrowing** - borrowed expression from foreign language (e.g. software),
9) **literal translation** - direct translation.

The theorist Vázquez-Ayora also includes **amplification** (extension of the text), **explicitation** (adding of explanatory information), **omission** and **compensation**. Another theorist Joseph L. Malone uses in addition terms **divergence** (you:ty/vy), **convergence** (ty/vy:you), **reduction** (Here I am:Zde), **condensation** (s modrým hřbetem: blue-backed), **diffusion** (tongue-heavy:mít těžký jazyk) and **reordering** (change of word order).

# 3 Characterization of scientific style (according to D. Knittlová)

Scientific style belongs to functional styles and has the informative function. Its aim is to provide the recipient accurate, clear and complete information from various fields of human activity. Scientific texts are characterized by the following features: accuracy, clarity, conceptuality, expertise and no emotions. Typical of texts of this style are technical terms and formulations. These texts are usually processed in written and monological form. The sentences are logically arranged and this contributes to better understanding and text comprehension. "The basic requirement of scientific style is the maximum usefulness and appropriateness of expression considering the function of scientific text, therefore temperance of expression." [7]

Scientific text is characterized by a simplified form due to keeping complete exactness of meaning. An important feature is vocabulary, because there is a specific layer of lexical units. Typical is the tendency to nominalization. Generally, in scientific style the subjective and expressive terms are not used. There is only a limited number of expressions and for that reason it leads to stereotyped selection of these words and comes frequent to repetition. In contrast to belles-lettres, these tendencies of stereotype and repetition contribute to the general transparency and text comprehension. Other resources, such as tables, graphs, etc. help its transparency and facilitate its perception.

Texts of scientific style are syntactically quite complicated. These are mainly complicated sentences, in which they are not used the expressive constructions. As mentioned above, the sentences have a logical structure and given composition which is to certain degree stereotypical. To achieve objectivity of the text the impersonal constructions are used, such as passive voice. Further connectors are used (called discourse markers), secondary prepositions and conjunctions for explication of relations between sentences. Other features are the condensation and schematization of text. Condensation of text is realized by infinitive constructions, participles and gerunds.

The occurrence of internationalization is another typical feature of scientific style. These are the elements that are to the given type of language inherently foreign.

International words are often used in scientific texts. The effort to adopt these loanwords was caused mainly by increasing contact between different languages. This process is related to the overall development of society and recently, as if this process is accelerated. The translator plays an important role in this process of acceptance.

We can distinguish between scientific style and popular scientific style. Popular scientific style is influenced by colloquial style and shares some features with journalistic style. This style also provides complicated topics in an interesting way and addresses general public. In contrast, scientific style addresses a small group of specialists. Also compared to scientific style is more graphic and descriptive. The sentences are rather shorter and do not contain almost any specific terminology. If there is a special term so it is explained in the text or demonstrated by an example. This style plays nowadays an important role, because it informs about the latest developments in various fields of science.

During the translation of scientific text is needed as perfect knowledge of both languages, such as in translation of belles-lettres. The translator of scientific text should have at least basic knowledge in the field, from which translates. In any case, the translator should work with an expert in a given field and consult the text with him in order to avoid misunderstanding, because "…what to the layman often seems quite correct and understandable in this context, the expert can reveal as false." [8]

# PRACTICAL PART

## 4 Translation of selected text

### 1 Objektová Technologie

Tato kniha se zabývá objektově orientovaným návrhem, UML (Unified Modelling Language - jednotný jazyk pro modelování) a programovacím jazykem Java. Snaží se prokázat, že Java aplikace, bez ohledu na to jak je malá, může mít prospěch z nějakého návrhu během její výstavby. Různá hlediska tohoto návrhu jsou zachycena a dokumentována pomocí UML.

V této knize se zaměříme zejména na střední prostor mezi objektově orientovaným návrhem a implementací. Existuje mnoho učebnic, které se výhradně zabývají programovacím jazykem Java (viz bibliografie). Tyto knihy nevěnují žádnou jednoznačnou pozornost otázce návrhu. Byl vydán mnohem menší počet knih o analýze a návrhu. Často se málo zabývají záležitostí realizace jejich návrhů. Zde se budeme snažit nabídnout propojení mezi nimi.

Výhodou tohoto přístupu je, že dochází k významnému posunu důrazu od podrobných programovacích problémů na vyšší rovinu analyzovat význam a přesnost návrhu. Kromě toho, když je vytvořeno schéma od návrhu až po realizaci jazyka, poznáváme vznikající a opakující se vzory a v důsledku toho se často změní velká část programovací činnosti do nepříjemné kódovací práce.

Tato úvodní kapitola poskytuje plán ke zbývající části knihy. Zde představíme podstatu objektově orientované práce s počítačem a poskytneme nezbytné úvodní podklady. Budeme zkoumat základy objektově orientovaného programování včetně modelování, analýzy, návrhu a implementace. Koncepty se vytvářejí u běžných zobrazení, ve kterých se soustředíme na zavedení slovní zásoby z oblasti objektově orientovaných systémů.

Počítačová technologie se od svého vzniku velmi rychle vyvinula. V současné době mají počítačově založené systémy vliv na velkou část našich životů v mnoha oblastech, jedná se o bankovnictví, zdravotnictví, rezervace letenek, vzdělávací a vojenské účely. Ty se vyznačují tím, že mají ve svém jádru velké množství

programového vybavení. Schopnosti těchto systémů jsou odvozeny ze složitých počítačových programů, které je ovládají.

Přestože lépe chápeme proces vývoje počítačového softwaru, mnohdy ho dodáme pozdě a překročíme rozpočet. Software často nedokáže dělat to, co uživatel požaduje, a je obtížné ho udržovat a upravovat. Tyto připomínky byly vždy aplikovány do průmyslu počítačového softwaru, a i když jsme zlepšili technologie na podporu procesu vývoje, ne zcela odpovídaly velikosti a složitosti současných systémů. Objektová technologie je považována za nejlepší pro splnění těchto výzev, nabízí nám prostředky ke zlepšení vývoje aplikací, spolehlivost a rozšiřitelnost.

## 1.1 Minulost

Během 90. let 20. století vstoupila objektová technologie do hlavního proudu v oblasti programování. V programovacích jazycích a v uvedení objektově orientovaných metod byly dva hlavní směry. Vývoj objektově orientovaných metod byl předmětem mnoha výzkumů jak organizací, tak jednotlivců. Významnými vůdci jsou Grady Booch (Booch 1991) a Jim Rumbaugh (Rumbaugh 1991). Každý z rozličných přístupů prosazovaných těmito a jinými měl nějakou výhodu, ale také měl vliv na roztříštění průmyslu.

V polovině 90. let Booch a pak Rumbaugh, později Ivor Jacobson, vytvořili organizaci Rational (http://www.Rational.com). Cílem bylo spojit jejich jednotlivé přístupy a příspěvky ostatních. Jejich snaha byla nabídnuta k veřejnému přezkoumání, čímž se získalo od průmyslu přijetí jedinečné jednotné objektově orientované (OO) notace. To poskytuje prostředky pro zachycení a zaznamenání různých prvků objektově orientované analýzy a návrhu (OOAD). Na konci 90. let zveřejnila organizace Rational řadu verzí UML. Následně v roce 1997 společnost Object Management Group (OMG) schválila UML jako standard nezávislý na výrobci.

### *1.1.1 Modelování*

Stejným způsobem, jakým architektův projekt představuje konstrukční podrobnosti pro budovu, UML umožňuje softwarovým modelům vybudování, zobrazení a řízení během analýzy a návrhu. *Modelování* je ověřený způsob používaný v různých disciplínách. Technické modely jsou například použity k návrhu a vývoji automobilů (automobilové strojírenství), letadel (letecké strojírenství) a mostů

(stavebnictví). Podobně meteorologové vyvinuli matematické modely k předpovědi počasí.

Modely jsou klíčové pro mnoho lidských aktivit. Poskytují plán pro nějaký výtvor, který chceme vyrobit nebo pochopit. Plán nabízí míru opakovatelnosti zajišťující standardizaci produktu. Je to stejně důležité, jako když si zákazník koupí software nebo řekněme domácí spotřebiče, jako například televizi nebo pračku.

Prostřednictvím modelu se snažíme poskytnout lepší pochopení systému ve vývoji. Modely napomáhají našemu pochopení obzvláště složitých systémů a pomáhají zajistit námi správné interpretování systému ve vývoji. Například automobiloví inženýři používají modely k navržení nových osobních vozů, které splňují řadu kritérií, včetně jejich aerodynamiky. Automobily mohou být vyrobeny ze sklolaminátu a mohou být testovány v aerodynamických tunelech. Podobný argument může být aplikován ve vývoji softwaru, přičemž model může zabezpečit splnění všech požadavků.

Model nám také umožňuje zhodnotit náš návrh podle kritérií, jako je bezpečnost nebo flexibilita. Podobně UML modely dovolí návrh aplikace vyhodnotit a posoudit před implementací. Provádět změny je mnohem jednodušší a méně nákladné, když jsou provedeny v raných fázích *životního cyklu softwaru*.

Modely nám pomáhají zachytit a zaznamenat naše rozhodnutí návrhu softwaru, zatímco postupujeme směrem k implementaci. To dokazuje, že je důležitým komunikačním prostředkem mezi členy vývojového týmu, stejně jako mezi nimi a zákazníkem. Členové vývojového týmu mohou diskutovat o svých návrzích s klientem, aby se ujistili, že plně rozumí jeho požadavkům.

Modely mohou být vrstvené a z toho důvodu poskytují různé úrovně detailu. Abstraktní modely softwaru opomíjejí velké množství jemného detailu a umožňují nám získat pohled vysoké úrovně na systém a jeho architekturu. Tyto pohledy nám dovolují soustředit se na různé části systému bez podrobné znalosti programového kódu. Opakované zdokonalování těchto modelů může být použito pro jejich posun ke konečnému kódu.

Architekt při projektování budovy často vytváří několik diagramů, které ji prezentují z různých perspektiv. Pochopitelně, jeden pohled je struktura budovy a je zásadní pro stavební firmy. Tentýž pohled je také důležitý pro zákazníka, protože odhaluje podrobnosti o ubytování, jeho uspořádání a přístupu ke schodištím, výtahům, atd. Nicméně, dodavatele elektřiny (nebo vody) více zajímá běh v elektrických

obvodech (dodávky vody a odpadní trubky), a jejich zásobování budovy energetickými společnostmi. Z tohoto důvodu by měl architekt sestrojit řadu plánů zdůrazňující tyto různé aspekty.

Podobným způsobem může softwarový architekt nabídnout celou řadu UML diagramů, které zobrazují systém z různých pohledů. Některé poskytují statický pohled na aplikaci s architektonickým uspořádáním objektů jako hlavní zaměření. Další UML modely kladou důraz na dynamické chování objektů a jejich vzájemné působení.

### 1.1.2 UML

UML definuje schematický zápis pro popisování výtvorů OOAD. Prostřednictvím UML můžeme znázornit, specifikovat, zkonstruovat a zdokumentovat naši softwarovou aplikaci. Jak se naše softwarové systémy stávají stále větší a složitější, musíme zvládnout tuto složitost a v určitém smyslu ji zjednodušit tak, abychom jí lépe porozuměli. Grafická vizualizace softwaru je mnohdy vhodnější než se to snažit pochopit v kódu programu.

Přezkoumáním našich modelů můžeme rozpoznat nedostatky našich návrhů, stejně jako příležitosti k jejich zlepšení. UML se chová jako upřesnění jazyka, ve kterém můžeme přesně a jednoznačně zachytit naše rozhodnutí při návrhu.

Nakonec můžeme z našich UML diagramů odvodit kód programovacího jazyka. Toto je označováno jako *progresivní inženýrství* - generování kódu z modelů UML. Jedná se o přístup, který prosazujeme prostřednictvím této učebnice. Modely jsou jádrem našich návrhů. Kód je výsledkem této modelovací činnosti a je sám o sobě dokumentem návrhu. Modely určují kód, který my nakonec vytváříme.

### 1.1.3 Analytické a návrhové modely

*Analytický model* použitý při vývoji softwaru se zaměřuje na dokumentaci různých aspektů problému z reálného světa, který my modelujeme. Ve vývoji objektově orientovaného systému by toto obvykle zahrnovalo rozpoznávání důležitých aplikačních objektů a aplikačních zpracování, které mají být provedeny.

Metody vývoje, které časově předešly objektovou technologii, jsou často úmyslně vymezeny mezi fázi analýzy a fázi návrhu. Běžně se také pro vývojový proces používá lineární nebo *vodopádový model*, ve kterém fáze návrhu následuje až poté, co byly dokončeny všechny analýzy. Kromě toho, tyto samostatné analýzy a fáze návrhu měly

často za následek konflikt mezi sebou, zejména tam, kde byly nasazeny různé modely a notace.

Objektově orientované metody jsou charakterizovány použitím stejných modelovacích konceptů v průběhu životního cyklu softwaru. Tímto způsobem se řešení, které se objevuje v průběhu analýzy, přenáší do návrhu a nakonec do kódu. Objekty identifikované v průběhu analýzy by měly být také přítomny v konečném kódu. To nabízí bezproblémové začlenění fází, jinak nenalezené s jinými přístupy. Návrhové modely rozšiřují analytické modely s dalšími detaily a možná i zavádějí další nízkoúrovňové systémové objekty potřebné při implementaci.

### *1.1.4 Proces vývoje*

UML je modelovací jazyk. Ten nemá představu o procesu vývoje, který musí doprovázet metodu. Slovník definuje metodu jako soustavnou nebo uspořádanou proceduru. Autoři UML porozuměli tomuto rozdílu a záměrně se snažili oddělit jazyk, použitý pro dokumentování návrhu softwaru, od postupu používaného k jeho vývoji. Uznali, že procesy jsou ovlivňovány mnoha aspekty, jako je povaha projektů a kultura organizací.

Objektově orientovaný návrh se obvykle provádí v rámci *iteračního procesu*. To je velmi důležité k zajištění toho, abychom se mohli znovu vrátit k dřívějšímu rozhodnutí, jsou-li nezbytné opravy nebo úpravy. Není to bezdůvodné. Konec konců, prvotní rozhodnutí o návrhu mohou vyžadovat revize, a to zejména v nových projektech nebo v těch, které jsou hůře srozumitelné pro vývojový tým. Iterační proces pokračuje, dokud není vyvinut celý systém.

U mnoha objektově orientovaných vývojářů je iterační proces také doprovázen *přírůstkovým* způsobem vývoje. Každý přírůstek zavádí nějakou přídavnou funkcionalitu k předchozímu stavu. Nový přírůstek často přidává jen malou funkci, takže můžeme plně otestovat jeho vliv na stávající systém a jeho architekturu.

Každá iterace musí být doprovázena cílem, který může být kontrolován. Jinak je zde nebezpečí, že proces propadne do *neukázněného hackerství*. Iterační přístup je dále rozšířen, když je zákazník úzce zapojen do vývoje systému. Každá nová iterace může být předložena zákazníkovi k získání zpětné vazby a k zajištění jeho aktivní účast na celém projektu.

## 1.2 Použití UML

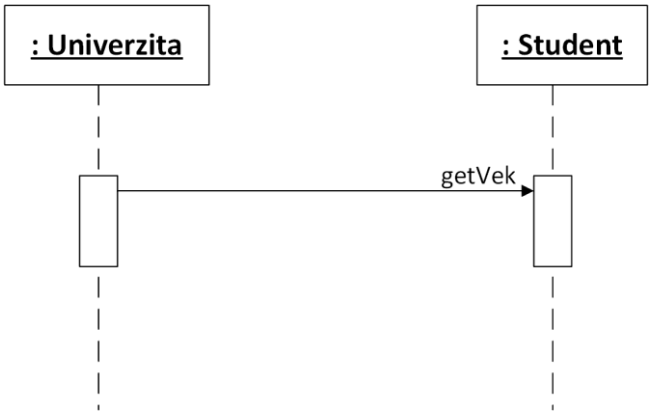
Další kapitola představuje podrobnou diskusi o UML. Zde jednoduše upevníme naše dřívější diskuze tím, že zvážíme důležitější prvky objektově orientovaného systému. To bude sloužit jako úvod do podrobnějších diskuzí, které budou následovat.

### *1.2.1 Objekty: kombinované služby a data*

Objektově orientovaný systém obsahuje řadu softwarových objektů, které se vzájemně ovlivňují pro dosažení cíle systému. Softwarové objekty obvykle napodobují skutečné objekty aplikační domény. V reálném světě mohou mít objekty fyzickou přítomnost nebo mohou představovat určitou dobře pochopenou koncepční entitu v aplikaci. Například v univerzitní aplikaci bychom mohli mít softwarové objekty, které představují studenty. Stejně tak můžeme mít softwarové objekty představující studijní programy na vysoké škole, i když nemají fyzickou existenci.

Objekty se vyznačují tím, že mají jak *stav*, tak *chování*. Stav objektu je informace o objektu samotném. Objekt student může mít například jméno, datum narození a univerzitní imatrikulační číslo. Stejně tak objekt studijní program může mít název programu, jeho trvání a jméno vedoucího programu. Chování objektu popisuje akce, které je objekt připraven vykonat. Mohli bychom se například zeptat objektu studijní program na dobu jeho trvání. Mohli bychom se zeptat objektu student na jeho věk. To by znamenalo, že objekt student provede výpočet na základě svého data narození a dnešního dne.

Chování objektu je popsáno souborem *operací*, které jsou připraveny se vykonat. Jeden objekt komunikuje s dalším tak, že jej požádá o provedení jedné ze svých ohlášených operací. Této interakce se dosáhne tím, že jeden objekt pošle *zprávu* druhému. První objekt je známý jako *objekt odesílatele* a druhý objekt je známý jako přijímač nebo *objekt příjemce*. Jediné zprávy, které může objekt přijmout, patří do množiny operací, které je možné přijmout. V UML jsou obvykle objekty a zasílání zpráv zachyceny *sekvenčním diagramem*, jak je znázorněno na obrázku 1.1. Zde je zobrazeno, jak objekt univerzita zasílá zprávu getVek na objekt student.




**Obrázek 1.1** *Zasílání zpráv v sekvenčním diagramu*

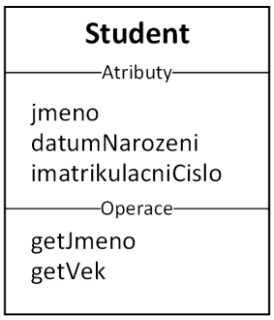
Když objekt obdrží zprávu, provede se nějaká akce. Tato akce je popsána *metodou*. Metoda jsou procesy jimiž se řídí přijímací objekt, když obsluhuje zprávu. Například, pokud objekt univerzita odešle zprávu objektu student s žádostí o jeho jméno, pak objekt student jednoduše odpoví odesílateli s jednou částí svého stavu, a to jménem. Nicméně, pokud objekt univerzita odešle objektu student zprávu s žádostí o jeho věk, pak metoda objektu student, která musí následovat, je propracovanější. Za prvé, musí získat dnešní datum. Toho může být dosaženo, když objekt student zašle zprávu nějakému objektu kalendář. Objekt student pak musí se svým datem narození a s datem dnešního dne provést některé komplexní aritmetické operace s daty pro výpočet jeho věku. V sekvenčním diagramu na obrázku 1.1 je toto zpracování zobrazeno jako *aktivace*, obdélník přiléhající k šipce zprávy.

Obrázek 1.1 také zahrnuje *šíření zprávy*. Když jeden objekt obdrží zprávu, často postupně odešle zprávu dalším objektům. Objekt univerzita odešle zprávu objektu student s dotazem na jeho věk. Ten, pro změnu, odešle zprávu některému objektu kalendář pro dotaz na dnešní datum. Tento příklad také dokazuje, že OO systém je mix objektů ovlivňujících se navzájem pro dosažení požadovaného cíle.

Univerzita by zpravidla měla mít velký počet studentů. Na rozdíl od skutečných studentů všechny objekty student projevují stejné chování a nesou o sobě stejné znalosti. Můžeme vymodelovat objekt student se jménem, datem narození a imatrikulačním číslem. Skutečné hodnoty stavu pro dva objekty student jsou pravděpodobně odlišné, jelikož univerzitní imatrikulační čísla jsou jedinečná. U velké univerzitní populace však můžeme očekávat dva nebo více studentů se stejným jménem nebo dva nebo více se stejným datem narození.

Nicméně oni všichni podléhají stejnému chování. Pokud bude jeden student požádán o sdělení svého věku zasláním vhodné zprávy, pak může být všem studentům poslána tato zpráva. Čím je to určeno? Všechny naše objekty student podporují jedinou abstrakci, kterou můžeme nazývat Student. Další abstrakce z této problémové oblasti mohou zahrnovat objekty Univerzita, ProgramStudia a StudijniVedouci. Odkazujeme se k abstrakci jako *třídě* objektu.

Třída je ve skutečnosti plán nebo šablona, která zcela popisuje abstrakci. Třída Student popisuje libovolný počet objektů student. Třída StudijniVedouci popisuje libovolný počet objektů studijní vedoucí. Třída popisuje informace, které má objekt, pro vyjádření svého stavu. Informační položky se nazývají *atributy* (někdy se jim také říká *parametry*). Třída také definuje chování takových objektů, zaznamenává operace, které mohou provádět, tj. zprávy, které mohou přijímat. Účel těchto operací je popsán svou metodou. Obrázek 1.2 ukazuje zjednodušený *diagram tříd* pro třídu Student.

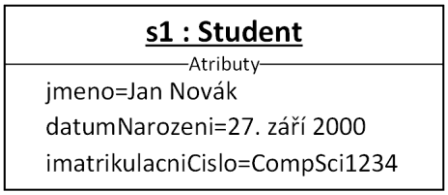



**Obrázek 1.2** *UML diagram tříd (zjednodušený) pro třídu Student*

Na tomto obrázku máme třídu Student se dvěma operacemi a třemi atributy. Každý objekt student, který tvoříme z této šablony, bude mít stav obsahující tři hodnoty atributů, tj. jmeno, datumNarozeni a imatrikulacniCislo. Dále, jakémukoliv objektu Student mohou být odeslány zprávy k získání jejich jména nebo věku, tj. getJmeno a getVek.

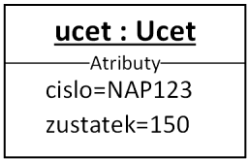
Obrázek 1.3 ukazuje, jak si v UML představujeme konkrétní příklad objektu z některé pojmenované třídy. Toto označujeme jako *instanci objektu* nebo jednoduše objekt. Horní část obrázku pojmenovává třídu, do které patří instance objektu, zde Student. Také označuje objekt nějakým identifikátorem, podle kterého můžeme odkazovat na tento objekt (s1). Spodní část představuje hodnoty atributů udržované instancí a znázorňuje její stav. Zde například tato konkrétní instance Student má atribut

jmeno s hodnotou Jan Novák jako součást jejího stavu. Takový prvek diagramu může být součástí mnohem většího *diagramu objektu* (nebo *diagramu spolupráce*), který si popíšeme pomocí UML (viz kapitola 2).




**Obrázek 1.3** *Instance objektu*

Dalším příkladem je rozmyslet si, jak můžeme modelovat bankovní účet. Účet může vykazovat různé chování, jako je odepsání nebo připsání určité peněžní částky, nebo si vyžádá aktuální stav na účtu. Toto chování vyvolává některé z možných operací účtu. Debetní a kreditní transakce dokládají částku obsaženou v transakci, mění zůstatek na účtu. Zůstatek, spolu s číslem účtu, musí být zachován každou instancí bankovního účtu. Každý příklad bankovního účtu nese své vlastní datové hodnoty pro tyto atributy (viz obrázek 1.4).




**Obrázek 1.4** *Instance bankovního účtu*

Pro modelování účtu jako objektu popíšeme jeho chování jako operace a jeho stav s atributy. Při práci systému je objekt účet požádán, aby provedl různé operace, změnou svých hodnot atributů podle potřeby tak, aby odrážely dopad akcí. Například na obrázku 1.4, operace odepsani aplikovaná na takový objekt účet, má za následek změnu hodnoty atributu zustatek.

Některé operace se používají k získání informací o objektu, zatímco ostatní mají nějaký vliv na stav objektu. Operace, které poskytují pouze informace o objektu, se označují jako *dotazovací operace*. Ptají se na nějaký stav informací, které jsou obsaženy v objektu. Operace k získání hodnoty zůstatku na účtu je tohoto typu. Operace, která provádí debetní transakce na objektu bankovní účet, změní aktuální zůstatek. Tato kategorie operace je popsána jako *transformační operace*. Ty mění jednu nebo

více hodnot atributu instance objektu. Obě operace se vztahují k hodnotám atributů objektu, všeobecně k *stavu* objektu. Transformační operace mají za následek změnu stavu, zatímco dotazovací operace obvykle nemají vliv na stav instance. V diagramu tříd na obrázku 1.5 můžeme rozpoznat operaci getZustatek jako operaci dotazovací, zatímco odepsani a pripsani jsou transformační operace.


**Obrázek 1.5** *Diagram tříd s třídou Ucet*

### *1.2.2 Objekty vytváří vynikající softwarové moduly*

Koncept objektu je jednoduchý a přesto velmi výkonný. Objekty vytváří ideální softwarové moduly. Každá instance objektu tvoří samostatnou entitu. Vše, co objekt zná, je vyjádřeno na základě jeho atributů, a vše, co je možné vykonat, je vyjádřeno jeho seznamem operací. Z tohoto důvodu jsou objekty popsány jako *vysoce soudržné*. Všechny vlastnosti objektu poskytují některé dobře vymezené chování pro určité abstrakce, které představují (*zapouzdření*).

Vezměme si automobil. Auto má různé ovládací prvky, které se používají k ovládání a řízení. Řadící páka se například používá pro změnu rychlostního stupně na ručně ovládaném vozidle nebo se používá pro zvolení převodu tam, kde je automatická převodovka. Auta mají obvykle tachometry, které ukazují rychlost vozidla. Další ovládací prvky zahrnují plynový a brzdový pedál.

Vnitřní součásti vozu jsou tvořeny mnoha mechanickými a elektronickými zařízeními, které jsou obsaženy uvnitř vozidla (obvykle pod kapotou motoru). Tato kovová kostra bezpečně odděluje řidiče od interního zařízení. Vzhledem k tomu, že řidič nemá přímý kontakt s těmito součástmi, není pravděpodobné, že poškodí sebe nebo auto. Vezměme si situaci, kdy namísto plynového pedálu ovládá řidič rychlost pomocí šroubováku na nějakém vnitřním regulačním šroubu. Plynový pedál má obvykle nějaký omezený rozsah pohybu, což řidiče omezuje v určité maximální rychlosti.

Bez tohoto omezení je možné, že úpravy provedené přímo regulačním šroubem mohou nastavit rychlost nad jeho maximální bezpečnou provozní úroveň a následně poškodit životně důležité součásti.

Stejné je to s izolováním řidiče od interního zařízení a pouze umožnit vyškolenému technikovi, aby získal přístup do součástek, čímž se dosáhne druhé výhody. Pokud je vozidlo porouchané, pak technik může vyměnit vadnou komponentu. Nicméně, změna byla uskutečněna bez jakékoliv operační změny. Řidič stále používá automobil naprosto stejným způsobem. Změnila se pouze oprava vadné komponenty.

Tyto myšlenky se také odráží v objektové technologii. Představují stejné výhody softwaru, jako je tomu na osobních automobilech. V softwarovém objektu jsou atributy skryty před uživatelem, ale mohou být nahrazeny bez ovlivnění softwaru uživatele, který se spoléhá na tuto abstrakci objektu (*skrývání informací*). V softwarovém objektu je chování určeno jeho operacemi, a nikoli jeho vlastní podobou. Operace řídí to, o co můžeme žádat softwarový objekt, aby to vykonal. Pokud dodáme nepřijatelnou hodnotu při vyvolání operace, pak může být účinek odepřen zajištěním správného stavu objektu. Například debetní operace aplikovaná na bankovní účet může být povolena pouze tehdy, pokud jsou k dispozici dostatečné finanční prostředky.

Jelikož objektové atributy a operace pouze definují své abstrakce a žádné jiné, obvykle vykazují *volnou vazbu* s jinými objekty. To je velmi žádoucí, protože *silná vazba* umožňuje softwarovým komponentám obtížnější porozumění, změnu nebo opravu. Například při stanovení objektové operace se nemusíme zajímat o žadatele (klient) této operace. Je nutné zvážit pouze vliv této operace na atributy přijímajícího objektu. Stejně tak, když klient žádá o službu, nemusí se zabývat o to, jak je této žádosti dosaženo u objektu příjemce.

### *1.2.3 Interakce objektu je vyjádřena jako zprávy*

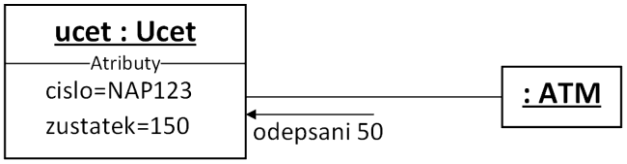
Obrázek 1.1 ukazuje, že objekty na sebe vzájemně působí zasláním zprávy od odesílatele, který požaduje po objektu příjemce provádět jednu ze svých ohlášených operací. Na obrázku 1.1 objekt Univerzita požaduje věk po objektu Student. Stejně tak objekt banka může poslat kreditní operace jednomu ze svých objektů účet. Zde je banka odesílací objekt a účet je přijímající objekt. Zpráva rozpoznává objekt příjemce a název operace, které mají být provedeny. Název zprávy představuje jednu z operací třídy, do

které patří příjemce. Jestliže zpráva vyžaduje jakékoli další údaje, jsou uvedeny jako *parametry zprávy*.

Chcete-li požádat objekt bankovní účet, aby se zabýval debetní transakcí některé peněžní částky, nějaký odesílací objekt, jako je bankomat (ATM), může poslat na objekt účet zprávu:

ucet odepsani 50

Zde, ucet je identifikátor objektu příjemce, což je nějaký objekt bankovní účet, odepsani je operace, o kterou se žádalo, aby byla vykonána, a 50 je skutečný parametr informující účet příjemce o částce podílející se na transakci. UML *diagram spolupráce* na obrázku 1.6 vykresluje toto předávání zpráv mezi objekty. V diagramu máme dva objekty: objekt Ucet s identifikátorem ucet a anonymní (bez identifikátoru) objekt ATM. Ten druhý odešle zprávu na objekt Ucet k provedení operace odepsani se skutečnou hodnotou parametru 50.




**Obrázek 1.6** *Zpráva z bankomatu (ATM) do objektu účet*

Když je zpráva přijata nějakým objektem příjemce, pak je akce provedena. Tato akce obvykle zahrnuje některé nebo všechny z hodnot atributů, které představují stav přijímacího objektu. Tato akce bude také používat jakékoliv parametry zprávy. Logika spojená s touto akcí je popsána metodou pro operaci. *Metoda* se vztahuje k *algoritmu*, který se použije při provedení operace. Metoda pro operaci odepsani použitá na objekt Ucet zahrnuje snížení hodnoty atributu zustatek o skutečnou hodnotu parametru zprávy 50.

Zaznamenali jsme, že transformační operace mají obvykle za následek změnu stavu na přijímajícím objektu, zatímco dotazovací operace od nich pouze požadují informace. Jediným prostředkem komunikace je zpráva odeslaná od odesílatele k příjemci. V případě dotazovací operace je pozorován sekundární tok informací. Zde, odesílatel očekává odpověď od příjemce ve formě *návratové hodnoty*. Transformační operace také občas poskytují návratové hodnoty, řekněme, zprávu pro

odesílatele, že určený úkol byl úspěšně dokončen. Ujistěte se, že rozpoznáte, že návratová hodnota není jiná zpráva.

Pozorujte také nesouměrnost konceptu zpracování zpráv. Objekt příjemce, když definuje svou operační logiku, se nezabývá objektem, který odesílá zprávu. Stejně tak se odesílatel nemusí zabývat tím, jak je prováděna operace příjemcem. Jak již bylo uvedeno v předchozí části, toto velmi napomáhá produkci vysoko kvalitních softwarových systémů oddělením těchto dvou záležitostí.

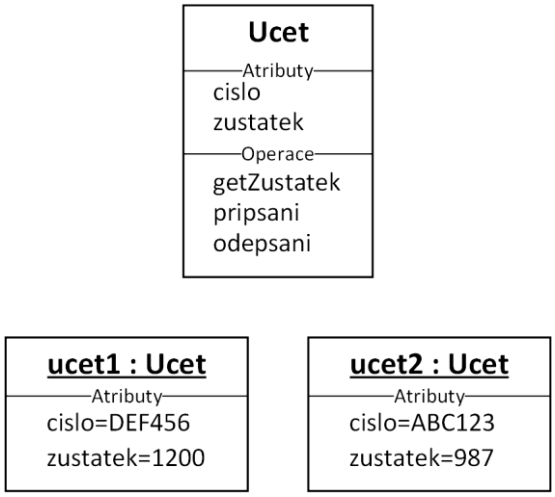
## 1.3 Třídy: soubor stejných objektů

Objektově orientovaný systém je charakterizován jako soubor interakčních objektů. Je proto běžné mít více než jeden objekt daného druhu. Například banka bude mít jistě celou řadu klientských účtů, z nichž každý provádí stejné akce a udržují stejný druh informací. Jednoduchá třída Ucet (např. obrázek 1.5) by mohla představovat celý soubor objektů účet (např. na obrázku 1.4). Třída obsahuje specifikaci a definici svých operací (metod) a jejich atributů. Skutečné účty jsou zastoupeny instancemi této třídy, z nichž každá má svůj vlastní jedinečný identifikátor (řekněme, ucet1, ucet2, ...). Každá instance obsahuje data, která představují svůj vlastní určitý stav. Pokud účet obdrží zprávu, aby provedl jednu ze svých operací, používá definici metody pro operaci danou ve své třídě a použije ji na vlastních hodnotách atributu.

Obrázek 1.7 ukazuje třídu Ucet a dvě instance této třídy. Instance mají identifikátory ucet1 a ucet2. Třída Ucet má tři služby poskytované operacemi odepsani, pripsani a getZustatek. Atributy udržované každou instancí této třídy mají své vlastní hodnoty pro cislo a zustatek. Například v instanci s identifikátorem ucet1 jsou tyto atributy popřípadě DEF456 a 1200. Zpráva:

```
ucet1 odepsani 50
```

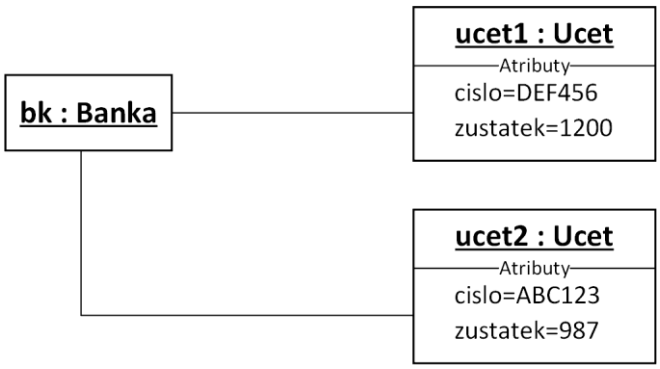
má za následek provedení metody pro operaci odepsani aplikovanou na objekt Ucet ucet1. Tato operace může být definována na základě odečtení hodnoty parametru zprávy 50 z hodnoty atributu zustatek v současnosti v držení objektu Ucet ucet1. Účinek této transformační operace vytváří změnu stavu v objektu ucet1 snižováním zustatek na 1150.



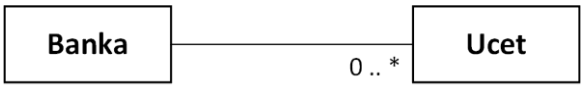

**Obrázek 1.7** *Třída Ucet a dvě instance*

Za normálních okolností objekty neexistují v izolaci. Vytvářejí vztahy mezi sebou a související objekty se zapojují do předávání zpráv. V diagramu objektů na obrázku 1.8 si ukážeme dva objekty Ucet, které se vztahují ke stejnému objektu Banka. Objekty Ucet spolu navzájem nesouvisejí. Takže objekt Banka může posílat zprávy jednomu nebo oběma objektům Ucet a ten druhý může posílat zprávy do objektu Banka. Je podstatné, protože neexistuje žádný vztah mezi objekty Ucet, a proto se nemohou zapojit do předávání zpráv.

Třídy a vztahy mezi nimi jsou modelovány v diagramu tříd na obrázku 1.9. Anotace 0..* označuje, že jeden objekt Banka může být ve vztahu s žádným (0) nebo s několika (*) objekty Ucet. Mnohem více budeme mluvit o těchto diagramech v následující kapitole. Diagram objektů na obrázku 1.8 je konkrétní příklad konfigurace objektů popsaných v tomto diagramu tříd. V podstatě jde o libovolný počet diagramů objektů, které jsou založeny na jediném diagramu tříd. Proto je diagram tříd abstraktním popisem všech možných konfigurací objektů.




**Obrázek 1.8** *Objekty a vztahy*




**Obrázek 1.9** *Diagram tříd se vztahem*

### *1.3.1 Specializace*

Objektově orientované modely mohou vést k vytvoření mnoha tříd objektů. Takové složité modely mohou být odůvodněny a zjednodušeny uspořádáním tříd do hierarchií. Hierarchie napomáhají při třídění typů objektových instancí představovaných mnoho třídami. Toto třídění vědomostí se vyskytuje v mnoha vědeckých a technických oborech a výrazně napomáhá při zjednodušování složitých systémů.

Mnoho lidí, bez ohledu na jejich konkrétní povahu, sdílejí společné rysy. Tyto vlastnosti zahrnují i informace o stavu a chování. Například jméno, adresa bydliště a číslo sociálního zabezpečení jsou údaje, které jsou sdíleny všemi lidmi. Stejně tak všichni sdílí stejné chování být schopni zeptat se osoby na jejich věk. Lidé mohou být dále zaměřeni jako zvláštní druh osoby, např. student. Kromě všech vlastností spojených s lidmi, mají i další vlastnosti typické pro to, být student, jako je například imatrikulační číslo. Vše, co se týká obyčejného člověka, platí i pro studenty a každá instance studenta je implicitně instancí osoby.

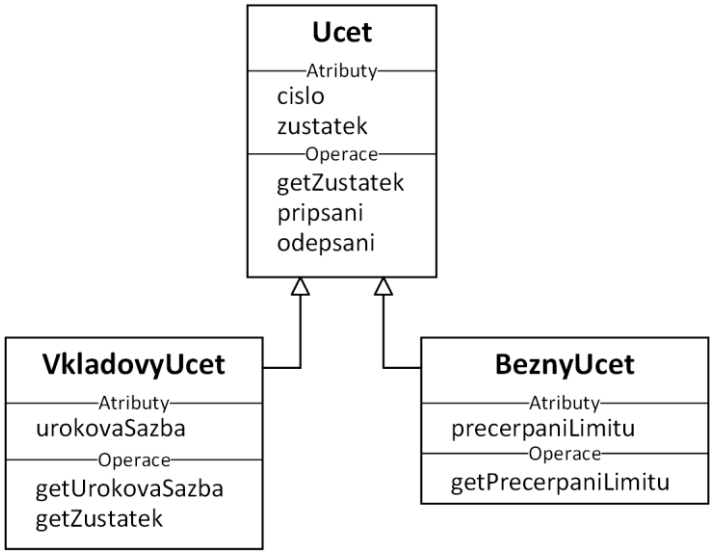
Třída Student je popisována jako *specializace* třídy Osoba. Naopak, třída Osoba je *zobecnění* třídy Student. Specializovaná třída Student *dědí* všechny vlastnosti svého zobecnění třídy Osoba. Proto, pokud každá osoba má jméno, pak ho mají i studenti. Pokud se můžeme ptát osoby na její věk, pak můžeme udělat totéž se studentem. Ve skutečnosti, každá operace, která může být použita na instanci Osoba,

může být použita i na instanci Student. Opak však není pravdou. Objekt Student může mít atributy a operace, které jsou pro něj typické, jako je například imatrikulační číslo. Jelikož třída Osoba je zobecněním třídy Student, pouze ty společné rysy jsou použitelné pro třídy Osoba. Z toho důvodu nemáme dovoleno ptát se osoby na její imatrikulační číslo, protože to platí pouze pro studenty.

Specializace je mechanismus, kterým je definována jedna třída jako speciální případ jiné třídy. Specializovaná třída zahrnuje všechny operace a atributy obecné třídy. Specializovaná třída dědí všechny vlastnosti (nebo charakteristiky) obecné třídy. Specializovaná třída může zavést dodatečné operace a atributy pro ni typické. Kromě operací zděděných si specializovaná třída může vybrat chování některé z nich a *nastavit znovu parametry*. Toto, jak brzy uvidíme, se používá, když má specializovaná třída více specifický způsob definování takového chování. Specializovaná třída je obecně známá jako *podtřída* a obecná třída je její *nadtřída*.

Specializace je často popisována jako *programování rozdílem*. Protože třída Student dědí od třídy Osoba, pak třída Student potřebuje pouze implementovat tyto rozdíly mezi sebou a zobecněnou třídou Osoba. Tak například, třída Student pouze zavádí další atributy typické pro studenty, řekněme, imatrikulační číslo. Dále prostřednictvím dědičnosti, specializovaná třída Student nemusí přeprogramovat zděděné operace. Rozdíl je jakákoliv další operace a jakákoliv nově definovaná operace. Podtřída má pak prospěch ze značného množství *opětovného použití kódu*.

V naší bankovní ilustraci třída Ucet by mohla být specializovaná do dvou podtříd BeznyUcet a VkladovyUcet. Specializace vztahu je znázorněna jako šipka směřující z podtřídy k nadtřídě, jak je uvedeno na obrázku 1.10. Každá zdědí obecné charakteristiky své společné nadtřídy. Buď si pak podtřída může přidat soubory operací a atributy nadtřídy, nebo může *předefinovat* chování jedné nebo více zděděných operací.




**Obrázek 1.10** *Dvě podtřídy bankovního účtu*

Na obrázku 1.10 má jakákoliv instance třídy BeznyUcet atributy cislo, zustatek a precerpaniLimitu. Poslední atribut je představen sám o sobě v třídě BeznyUcet, zatímco ostatní atributy jsou zděděny od nadtřídy Ucet. Stejně tak každá instance BeznyUcet může reagovat na zprávy odepsani, pripsani, getZustatek a getPrecerpaniUctu. Opět platí, že první tři operace se dědí od nadtřídy a poslední je definována pro třídu BeznyUcet. Podobné uspořádání platí pro třídu VkladovyUcet, ve které jsou atributy instance cislo, zustatek (obě zděděné) a urokovaSazba (uváděná třídou). Operace jsou odepsani, pripsani (obě zděděné), getUrokovaSazba (definována třídou) a getZustatek (znovu definována).

Obnovení definice operace v podtřídě umožňuje specializovaná implementace metody. Například operaci getZustatek ve třídě Ucet lze definovat tak, že jednoduše vrátí současnou hodnotu atributu zustatek. V podtřídě VkladovyUcet znovuobjevení operace getZustatek označuje předefinování, které by se mohlo například zabývat narůstajícím úrokem.

### *1.3.2 Polymorfismus*

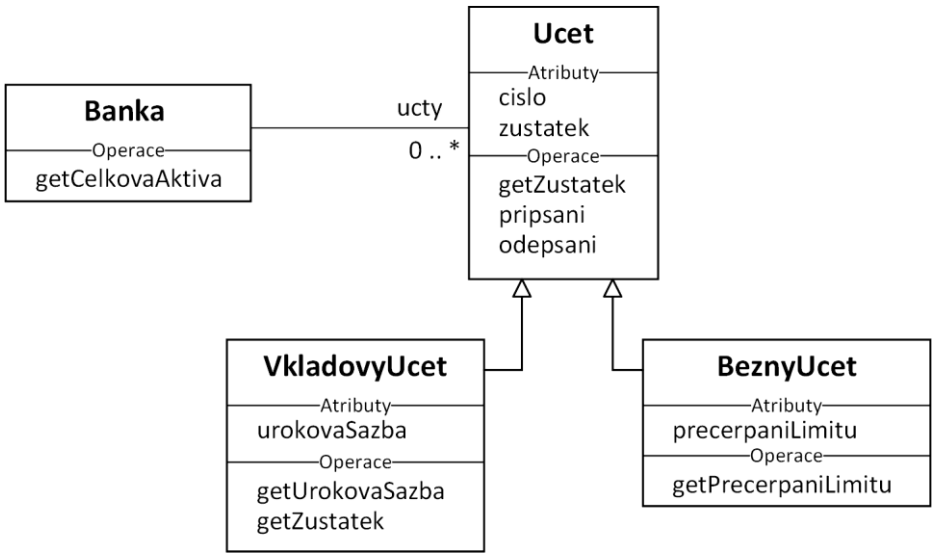
Pomocí dědičnosti je vytvořena jedna třída jako specializace existující třídy, dědí všechny vlastnosti této existující třídy. To se ukázalo být obzvláště důležitým konceptem, který podporuje znovu použitelnost stávajícího kódu. Dědičnost také vede k pojmům *polymorfismu* a *dynamické vazby*. Tyto dodatečné koncepty poskytují podporu,

pomocí které softwarové systémy mohou být upravené tak, aby přizpůsobila změny své specifikace.

Slovníková definice pro polymorfismus je "mající mnoho podob". Definice třídy pro VkladovyUcet deklaruje explicitní specializaci z třídy Ucet a odhaluje, že VkladovyUcet je Ucet s dalšími atributy, operacemi a obnovenými operacemi. Proto instance VkladovyUcet může být nahrazena instancí Ucet. Je to přípustné, protože instanci třídy VkladovyUcet může být zaslána stejná zpráva jako instanci třídy Ucet. Stejně tak, instance BeznyUcet může být použita tam, kde se očekává instance Ucet. To znamená, že například objekt Banka může být uveden mnoha objekty Ucet, které jsou s ním spojeny. Objekt Banka se nemusí zajímat, zda se jedná o objekty BeznyUcet nebo objekty VkladovyUcet. Všechny jsou typem objektu Ucet.

Tento přístup je zcela odlišný od toho, který se používá v běžných systémech, kde je nezbytné naplnit kód složitým výběrem příkazů k identifikaci typu účtu a pak ke spuštění nějaké vhodné logiky. V těchto systémech záleží zcela na programátorovi, jaký typ účtu vybere. V objektově orientovaných systémech je odpovědnost za tento výběr dána programovacímu prostředí.

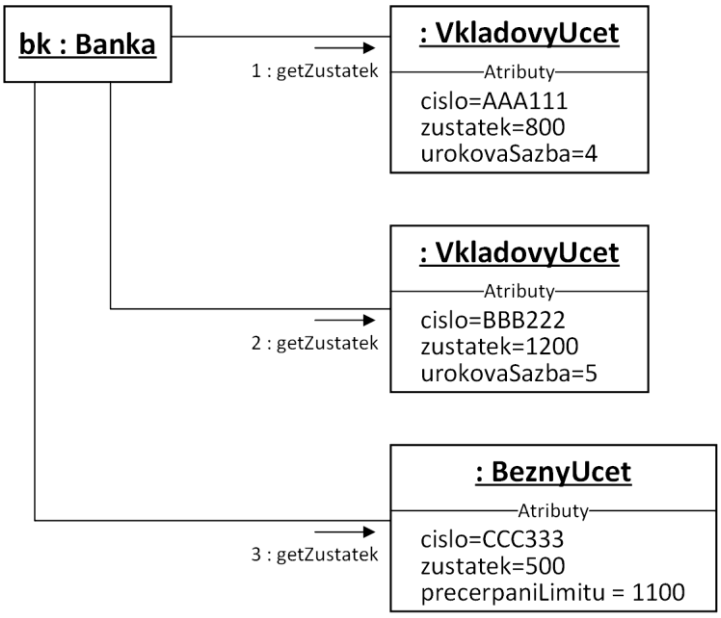
Obrázek 1.11 představuje diagram tříd, v nichž počet účtů je držený nebo udržovaný bankou. Jednotlivá instance třídy Banka je odpovědná za žádné nebo více instancí různých druhů bankovních účtů, z nichž některé jsou VkladoveUcty a některé BezneUcty. Označená čára je *spojení* ukazující vztah jeden k několika mezi třídou Banka a obecnou třídou Ucet. Tento typ modelu objektů je předmětem této knihy a je formálně představen v následující kapitole.




**Obrázek 1.11** *Diagram tříd* Banka/Ucet

Když objekt Banka odešle zprávu getZustatek na každý z mnoha objektů Ucet, pak se použije pro tuto operaci definice ve třídě Ucet. Některé účty budou samozřejmě úročitelné VkladoveUcty, které díky předefinování metody getZustatek mají odlišný způsob výpočtu úrokové částky. K získání správné volby metody odložíme výběr metody k provedení až *za běhu* použitím mechanismu známého jako *dynamická vazba*. Toto je provedeno záznamem, že *polymorfismus* je nutný k operaci getZustatek. Pokud operace getZustatek ve třídě Ucet je polymorfní, pak, když je zpráva odeslána na každou instanci účtu, tak odpovídající definice operace je provedena podle třídy přijímacího objektu. Efektivní je, když přijímající objekt ví, do které třídy patří a provede příslušnou metodu.

Proto, když je zpráva getZustatek přijata instancí VkladovyUcet, pak je provedena nově definovaná verze metody z této třídy. Když objekt BeznyUcet obdrží stejnou zprávu, potom protože tato třída nepředefinuje operaci, vykonaná metoda je definována a zděděna z nadtřídy Ucet. UML diagram spolupráce na obrázku 1.12 ukazuje, jak objekt Banka určuje hodnotu svých celkových aktiv. Každý účet je buď instance VkladovyUcet a nebo instance BeznyUcet. Banka si není vědoma této skutečnosti a jednoduše odešle zprávu getZustatek každému. Když jeden objekt BeznyUcet obdrží zprávu getZustatek, jednoduše provede metodu zděděnou z nadtřídy Ucet. Dva objekty VkladovyUcet používají tuto nově definovanou metodu ve své podtřídě.




**Obrázek 1.12** *Provedení dynamické vazby*

Polymorfismus přispívá k udržování programu. Pokud bychom měli přidat nový typ účtu do systému, řekněme třídu SporiciUcet jako specializaci třídy Ucet, mohl by implementovat svou vlastní operaci getZustatek. Třída Banka nemusí vědět o tomto rozšíření. Opět se odešle zpráva getZustatek každému účtu. Je-li konkrétní instance náhodou z této nové třídy, zvolí si svou vlastní definici pro tuto operaci.

## 1.4 Nástroje

Moderní vývoj softwaru dnes obvykle probíhá za podpory nástrojů pro vývoj softwaru. Jsou obecně popisovány jako nástroje *počítačem podporovaného softwarového inženýrství* (Computer Aided Software Engineering - CASE). Mnoho komerčních CASE nástrojů jsou velké a komplexní softwarové systémy, které podporují mnoho fází vývojového procesu softwaru. Jsou často známé jako upper-case nástroje kvůli jejich podpoře pro většinu aspektů procesu. Pro srovnání, lower-case nástroje poskytují menší podporu, ale obvykle vyžadují mnohem kratší průběh učení.

Na podporu čtenáře během zbývající části této učebnice, autoři dali k dispozici lower-case nástroj ROME (viz přílohy A a B). Takto může čtenář sledovat obsah knihy a má přístup k nástroji CASE na podporu různých částí diskuse. Zatímco upper-case nástroje se snaží automatizovat činnosti, zajistit soulad napříč modely, poskytovat různé zprávy o řízení, atd., ROME neobsahuje tyto funkce, aby zajistil mnohem jednodušší

nástroj k fungování. Na konci této studie však předpokládáme, že čtenář by měl být schopen postoupit k pokročilejší komerčním nástrojům. Podívejte se například na webové stránky http://www.rational.com, http://www.togethersoft.com.

Modelovací nástroj ROME byl používán v celé této knize k vytvoření mnoha zobrazených UML diagramů. Aktuální verze ROME podporuje většinu diagramů popsaných v UML. Dále, program pro vytváření diagramu tříd v ROME se používá pro generování kódu v jazyce Java.

Podívejte se na část označenou "Distribuce softwaru" v předmluvě pro podrobnosti o tom, jak získat a nainstalovat dodaný software.

## 1.5 Shrnutí

1. Jednotný jazyk pro modelování (Unified Modelling Language), UML, je mezinárodně dohodnuté označení pro zaznamenávání různých prvků objektově orientované analýzy a návrhu. UML definuje řadu pohledů na systém prostřednictvím různých diagramů, jako jsou třídy a diagramy spolupráce.
2. UML musí být rozšířeno pomocí procesu řízení vývoje softwaru. V OOAD jsou použity stejné koncepce modelování v celém vývojovém procesu softwaru.
3. Objektově orientovaný systém je charakterizován jako soubor objektů komunikace.
4. Objekt je souborem operací společně se stavem, který zachovává objekt mezi vyvoláním některé z jeho operací. Transformační operace mají za následek změnu tohoto stavu, zatímco dotazovací operace podávají zprávu o stavu.
5. Instance objektu je konkrétní příklad nějakého objektu z pojmenované třídy a může být zobrazena v UML diagramu objektu. Třída je plán nebo šablona popisující libovolný počet takových instancí a je uvedena v diagramu tříd.
6. Objekty komunikují pomocí předávání zpráv, jsou zobrazeny buď v UML diagramu spolupráce nebo v diagramu sekvencí. Jeden objekt zašle jinému objektu zprávu, která vyvolá některou z operací příjemce. Zpráva je vázána na definici operace dané své třídě nebo nadtřídě.
7. Třídy mohou být zařazeny do hierarchie počínaje od obecného a vedoucí ke konkrétnějšímu. Podtřída je specializace své přímé nadtřídy. Podtřída dědí všechny vlastnosti své nadtřídy. Mohou se přidat další funkce a obnovené operace. Instance podtřídy je instancí své nadtřídy a může být kdykoli nahrazena za instanci té druhé.

8. Dědičnost také vede k pojmům polymorfismu a dynamické vazbě. Dynamicky vázaná, polymorfní zpráva odeslaná objektu se váže k definici operace ve třídě, do které patří objekt.

# 5 Commentary

## 5.1 Macroanalysis

This text is written in scientific style. Characteristic features are evident and are described in the theoretical part, which deals with scientific style. The source of the text is a book called "Object-oriented design with UML and Java". Authors of this book are Kenneth Barclay and John Savage. They have expert knowledge in the field of information technology and programming.

The function of the text is informative and expression is formal and objective. It provides information to the public which is interested in the topic and try to improve their knowledge. The recipient of this text is specific group of people with an interest in programming and computers. It could be, for example, students who use this book to supplement the knowledge of the field. They are mostly people who already have some knowledge and experience in this field.

The text is characterized by accuracy, clarity of expression and condensation. The condensation in the source language is expressed by dependent clauses in the target language. Between individual parts of the text is sequenced logical system. The text is clearly divided into chapters, subchapters, and lastly into paragraphs. The length of sentences is adequate. A part of the text are pictures with descriptions which help its understanding. These pictures are translated and inserted into the target text. The author tried as much as possible to emulate the source text.

### *5.1.1 Lexical analysis*

**Terminology:** the text is typical for using technical terms, which are explained within the text. But the reader must have as well some knowledge in field of IT. E.g. *forward engineering*, *hacking*, *encapsulation* and others, which are mentioned in the glossary.

**Hyphenated compounds:** a compound word is a combination of two or more words that function as a single unit of meaning. These words condense the text. E.g. *object-oriented, computer-based, fine-grained, low-level, one-to-many*.

**Abbreviations:** they are explained in the text. E.g. *UML (Unified Modelling Language), OOAD (object-oriented analysis and design), OMG (Object Management Group), OO (object-oriented), ATM (automated teller machine)*.

**Linking words:** they perform different functions and are placed at the beginning of a sentence. E.g. *also, however, in addition, therefore, thus.*
**Numbers**: they are used in the text for the expression of the year (e.g. *1990s, 1991*) and then in pictures and examples of attributes (e.g. *NAP123, debit 50, DEF456*).
**Proper nouns:** some computer scientist are mentioned in the text. E.g. *Grady Booch, Jim Rumbaugh, Ivor Jacobson*

### *5.1.2 Gramatical analysis*

Thanks to above mentioned language features of the text, it is clear that the text is written in scientific style. The text has informative function, processes a technical topic, is mainly designed for specific group of people and the reader and also the author are required to have some knowledge in this field. The text uses technical terminology and contains no expressive words. The text is typical for its condensation (hyphenated compounds, gerunds, participles), objectivity, impersonal constructions (passive voice) and linking words.

**Active voice:** is used in the text but it is not frequent as passive voice. The subject performs the action expressed in the verb. E.g. *a model also **permits** us to evaluate our design, an architect **constructs** a number of diagrams, a message **identifies** the recipient.*
**Passive voice:** prevails in the text. Is used to express information in an impersonal way and is used when we can highlight the action. E.g. *design **are captured** and **documented**, models to **be constructed**, **viewed** and **manipulated**, notations **were deployed**.*
**Personal pronoun:** the using of active form with personal pronoun "we" refers to the authors of the book. E.g. *we seek, we can identify, we can fully test, we concentrate on, we aim.*
**Tenses:**

- **Present simple** prevails in the text. It refers to the fact in the present period of time. E.g. *objects **make** ideal software modules, the class **contains** the specification and definition, these additional concepts **provide** support.*
- **Present perfect** is also used in the text. It refers to action that continues to the present. E.g. *these remarks **have** always **applied** to the computer software*

*industry, and while we **have improved** the technologies, the authors **have made** available a lower-case tool ROME.*

- **Past simple** refers to completed actions. E.g. *object technology **entered** the mainstream, the various approaches **promoted**, Object Management Group **approved** the UML.*

It is used to condense the text:

- **Present participles:** e.g. ***emerging** patterns, **programming** languag*e, ***modelling** activity, **existing** system, **sending** a message.*
- **Gerunds:** e.g. ***realizing** their designs, **analysing** the meaning, **developing** computer software, for **describing** the artefacts.*
- **Infinitive constructions:** e.g. *we seek **to offer** a bridge, is difficult **to maintain**, the aim was **to combine** their approaches, a model can be used **to ensure** all requirements, object can be sent messages **to obtain** their name.*

The text contains **no direct speech**.

## 5.2 Microanalysis

The author introduces the most interesting and typical examples how she proceeded while translating. She focuses mainly on the translation of terms, because they are the most important to understanding of the text.

In some sentences was necessary to use the translation using dependent clause due to the different grammatical structure of the source and target language. Most often it was in the case of the infinitive construction, as well as participle and gerund.

***The class describes the information an object holds to represent its state***.
*Třída popisuje informace, které má objekt, pro vyjádření svého stavu.*
***During the execution of a system, an account object is requested to carry out its various operations, changing its attribute values as needed to reflect the effect of its actions.***
*Při práci systému je objekt účet požádán, aby provedl různé operace, změnou svých hodnot atributů podle potřeby tak, aby odrážely dopad akcí.*
**object-oriented design:** *objektově orientovaný návrh* → condensation at translation of hyphenated compound.
**computer software:** *počítačový software* → the word *software* is borrowed from English and it is used as anglicism.
**in the mid-1990s:** *v polovině 90. let* → condensation at translation of hyphenated compound.
**vendor-neutral standard:** *standard nezávislý na výrobci* → condensation, it is author´s own solution.
**wind tunnels:** *aerodynamické tunely* → instead of the word *větrný* was used *aerodynamický* based on the context.
**diagrammatic notation:** *schematický zápis* → it is generally used.
**forward engineering:** *progresivní inženýrství* → the word *progresivní* is author´s own solution and it is based on the fact, that *forward* means advanced or innovative.
**analysis and design models:** *analytické a návrhové modely* → it is common naming for this types of models.
**after all:** *konec konců* → this can be translated also as *potom všem* but the author chose the not literal expression.

**undisciplined hacking:** *neukázněné hackerství* → the English word *hacking* can be translated as *hackerství* or may not be translated (borrowing).

**feedback:** *feedback* → borrowing, it can be also translated as *zpětná vazba*.

**message propagation:** *šíření zprávy* → this expression is generally used.

**…we need not concern ourselves with the requester (client) of that operation.:** *...nemusíme se zajímat o žadatele (klient) této operace.* → the word *ourselves* expresses here the reflexive pronoun "*se*".

**…does not concern itself with the object …:** *...se nezabývá objektem ...* → the word *itself* expresses the reflexive pronoun "*se*".

**programming by difference:** *programování rozdílem* → this expression is author´s own solution.

**"having many forms":** *mající mnoho podob* → it is literal translation.

**one-to-many:** *jeden k několika;* **interest-bearing:** *úročitelný;* **run-time:** *doba běhu* → condensation.

**redefine:** *nastavit znovu parametry* → it is used amplification or it can be translated as *předefinovat*.

**Computer Aided Software Engineering (CASE) tools:** *nástroje počítačem podporovaného softwarového inženýrství (CASE)* → this translation is based on the searched expression on the internet, but commonly the acronym *CASE* is used.

**upper-case:** *upper-case* → borrowing from English and it is used as anglicism (it is the same with the word *lower-case*).

**Here, for example, this particular Student instance has theName attribute with the value Ken Barclay…:** *Zde například tato konkrétní instance Student má atribut jmeno s hodnotou Jan Novák...* → for the translation of the name *Ken Barclay* the typically Czech name *Jan Novák* has been used.

### *5.2.1 Using Java naming conventions*

A naming convention is a set of rules for choosing the character sequence to be used for identifiers (e.g. class, method, attribute, etc.). Java programmers can have different styles to the way they program. By using Java naming conventions they make their code easier to read for themselves and for other programmers.

In the source text these cases were used:

- **CamelCase** is where each new word begins with a capital letter and it is used for classes.
  **University, ProgrammeOfStudy, Tutor, Person:** *Univerzita, ProgramStudia, StudijniVedouci, Osoba*
  **CurrentAccount, DepositAccount, SavingAccount:** *BeznyUcet, VkladovyUcet, SporiciUcet*
- **Mixed case** is the same as CamelCase except the first letter of the name is in lowercase and it is used for attributes and methods.
  **theName, theDateOfBirth, theMatriculationNumber, theBalance:** *jmeno, datumNarozeni, imatrikulacniCislo, zustatek*
  **getName, getAge, getBalance, getOverdraftLimit, getInterestRate:** *getJmeno, getVek, getZustatek, getPrecerpaniUctu, getUrokovaSazba*
  **debit, credit:** *odepsani, pripsani*
  **acc1, acc2:** *ucet1, ucet2*

For obtaining the values of data items so-called "getters" are used, i.e. methods having in its name the word get. They are not translated into Czech. For example **getName** is translated as *getJmeno*, etc.

# 6 Glossary

Glossary is created mostly from the marked words in the original text (italic). These are mainly technical terms related to this topic. For searching the meanings of these words the website The Free Dictionary was used. Sometimes it was not easy to find an exact equivalent in the Czech language.

**Algorithm:** a step-by-step procedure for calculations.

- algoritmus

**Analysis model:** it describes the structure of the system or application that you are modelling. It consists of class diagrams and sequence diagrams.

- analytický model

**Association:** defines a relationship between classes of objects that allows one object instance to cause another to perform an action on its behalf.

- spojení

**Attribute:** a specification that defines a property of an object, element, or file.

- atribut

**Behaviour:** what the objects do, e.g. Student attends a course "Java for beginners".

- chování

**Class:** is an extensible blueprint or template for creating objects.

- třída

**Class diagram:** a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among the classes.

- diagram tříd

**Code reuse:** the use of existing software, or software knowledge, to build new software.

- opětovné použití kódu

**Cohesion:** refers to the degree to which the elements of a module belong together. It is usually described as “high cohesion“ or “low cohesion“.

- soudržnost

**Collaboration diagram:** describes interactions among objects in terms of sequenced messages.

- diagram spolupráce

**Design model:** it builds on the analysis model by describing, in greater detail, the structure of the system and how the system will be implemented.

- návrhový model

**Dynamic binding:** a computer programming mechanism in which the method being called upon an object is looked up by name at runtime.

- dynamická vazba

**Encapsulation:** a language mechanism for restricting access to some of the object's components. The term encapsulation is often used interchangeably with information hiding (see also Information hiding).

- zapouzdření

**Forward engineering:** the process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.

- progresivní inženýrství

**Hacking:** use of a computer system without a specific, constructive purpose, or without proper authorization.

- hackerství

**Incremental style:** to reduce project risks by dividing the project into smaller segments and simplify the possibility of introducing changes during the development process.

- přírůstkový způsob

**Information hiding:** the ability to prevent certain aspects of a class or software component from being accessible to its clients. The term encapsulation is often used interchangeably with information hiding (see also Encapsulation).

- skrývání informací

**Inheritance:** refers to the ability of one class (subclass) to inherit the identical functionality of another class (superclass), and then add new functionality of its own.

- dědičnost

**Iterative process:** a process for calculating a desired result by means of a repeated cycle of operations, which comes closer and closer to the desired result.

- iterační proces

**Loose coupling:** loosely coupled system is one in which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components.

- volná vazba

**Message:** it defines a specific kind of communication between instances in an interaction.

- zpráva

**Method:** a piece of code associated with a class or object to perform a task

- metoda

**Modelling:** or simulation, the scientific method often used to describe reality, to make or construct a model of.

- modelování, simulace

**Object diagram:** a diagram that shows a complete or partial view of the structure of a modeled system at a specific time.

- diagram objektu

**Object instance:** a specific realization of any object. Formally "instance" is synonymous with "object", as they are each a particular value (realization), and these may be called an instance object.

- instance objektu

**Operation:** a process or procedure that obtains a unique result from any permissible combination of operands.

- operace

**Run-time:** the period during which a computer program is executing.

- doba běhu

**Sequence diagram:** kind of interaction diagram that shows how processes operate with one another and in what order. It shows object interactions arranged in time sequence.

- sekvenční diagram

**State:** what the objects have, e.g. Student have a first name, last name, age, etc.

- stav

**Waterfall model:** a sequential design process, often used in software development processes, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of conception, initiation, analysis, design, construction, testing, implementation, and maintenance.

- vodopádový model

# 7 Anglicisms

Anglicism is language element borrowed from English into another language. These expressions occur in the terminology of some fields, into Czech penetrated in several waves and some of them gradually became domesticated. In some fields, the development reached the creation of steady slang that combines Czech and English. English gains ground in the Czech language more and more. Anglicisms are most often used by young generation.

## 7.1 Development

English expressions began to spread in the world already in the 19th century. The Industrial Revolution brought along the technical terms (e.g. *tramway*, later *tramvaj*). In the 20th century began to appear mainly terms from sports (e.g. *fotbal*, *volejbal*). There are also words from other areas (e.g. *víkend*, *gaučˇ*). After 1990 English words most significantly affected the area of informatics and computer technology, advertising, business and management. The most widely used are denotations of firms and commercial, cultural and sports centers with English names. Original Czech names are in them incorporated according to English grammar and spelling (e.g. *Cinema City*, *Sazka Arena*). Typical for this phenomenon is that the nominal tight/restrictive apposition placed according to Czech grammar after noun is replaced by indeclinable attribute placed before the developed name (*v Sazka Areně* instead of *Aréně Sazka*). This phenomenon was demonstrated in Czech already in the 1930s, such as the name of the theatre *Rokoko kabaret*.

Some people consider flood of anglicisms as a negative phenomenon. But not every borrowed word from English is harmful for Czech. For example, in the technical areas help these expressions in the absence of Czech equivalents (computer terms such as *software* or *upgrade*). In some cases it is also a question of trendiness when someone uses rather words *meeting* or *tým* instead of Czech words *schůzka* or *skupina*.

Czech vocabulary in the field of information technology is enriched by a wide range of vocabulary from English. Some of the words have become a part of the Czech vocabulary and are commonly used in everyday communication (e.g. *e-mail*, *SMS*). This process of making words Czech still continues, although in most cases the

expressions leave both spelling and pronunciation. In this area are a lot of scientific names and terms and those are understood mainly by people who work with computers. Czech programmers are accustomed to using English terms.

## 7.2 Formal adaptation

The main problem in adopting of anglicisms is for Czech the unusual difference between sound and graphical form. These forms coincide only by a small number of terms (e.g. *film, fit, set*). It is therefore questionable whether make the borrowed English words Czech or keep their original form. Most of anglicisms adapt grammatically in Czech after a certain period of operation in the original graphical form. There are two ways of creation of a new spelling form:

a) basis is the original graphical form that adapts the pronunciation (*basketbal, tramvaj*),
b) basis is the pronunciation that adapts the graphical form (*byznys, víkend*).

There are also irregularities that indicate the individuality of this process (e.g. *kečup, džungle*) and also ambiguous relation between Czech and English phonemes (e.g. *teč, gól*). Systematism of scientific names may be disturbed by double result of adaptation of one morpheme, which is caused by different time of adoption (e.g. *fotbal, volejbal*). In contrast, the new names permeated to us later with the original English pronunciation (e.g. *baseball, softball*). A major role in the adaptation plays effort to achieve formal similarity with native words, especially in the ending. This leads to the fact that borrowed lexical units may be incorporated into the morphological system of Czech.

Currently is the problem of double spelling in newly borrowed words (e.g. *leader*/*lídr, marketing*/*marketink*), but also by words that are in the language for a long time and their Czech form asserts slowly (*toast*/*toust, handicap*/*hendikep*). The withdrawal of Czech form in doublets (e.g. *jazz*/*džez*) or recovery of form *manager* next to *manažer* reflects that the development does not direct only to making Czech, but that here also acts opposite tendency. In connection with the increasing international prestige of English, it will be shown whether this is a temporary phenomenon or not.

Some words have no Czech form at all. The reasons are formal and functional, but it is difficult to determine accurate rules. There are mainly more words and

composites (e.g. *fair play, happy end, play-off*) and lexical units with unusual graphical form (*tie-break, interview, outsider*).

Formal adaptation is one of many problems that are associated with the using of anglicisms. The functionality of borrowed words, their style evaluation, relationship to the Czech equivalent and others represent extensive problem that should have separate processing.

## 7.3 Examples of anglicisms

These anglicisms occur in the selected text.

**Table 1:** *Anglicisms in the selected text*

| in Czech | in English |
|---|---|
| feedback | feedback |
| hacking/hackerství | hacking |
| lower-case | lower-case |
| software | software |
| upper-case | upper-case |
| website | website |

Due to the lack of examples in the selected text the other examples from this area are mentioned. These terms were discussed with the relevant people in this field.

**Table 2:** *Another examples of anglicisms*

| in Czech | in English |
|---|---|
| bit | bit (binary digit) |
| bluetooth | bluetooth |
| buffer | buffer |
| byte | byte |
| e-mail | e-mail |
| firewall | firewall |
| hacker | hacker |
| hardware | hardware |
| hashování/hešování | hashing |
| internet | internet |
| laptop | laptop |
| login | login |
| offline | offline |
| online | online |
| paket | packet |
| reset | reset |
| roaming | roaming |
| server | server |
| skener | scanner |
| upgrade | upgrade |

There are a lot of international words in the text (i.e. internationalisms) of other origin than English is. These are words or expressions occurring in several languages and having in them the same or similar meaning. Influenced by English they are adopted into Czech without considering the other version of the word.

**Table 3:** *Internationalisms in the selected text*

| in Czech | in English |
|---|---|
| architektura | architecture |
| definovat | to define |
| dokumentovat | to document |
| implementace | implementation |
| instalovat | to install |
| instance | instance |
| iterace | iteration |
| komponenta | component |
| konstruovat | to construct |
| notace | notations |

# 8 Conclusion

The main objective of this thesis was to create a translation from the field of IT from English into Czech with a commentary and glossary. As mentioned in the introduction, the thesis is divided into theoretical and practical part. The first part was presented the theory of translation, a brief description of scientific style in comparison with popular scientific style.

All theoretical knowledge was applied in the practical part, in which the translation was introduced. The biggest problems during translating of this text lie in a large number of technical terms from the field of information technology, which have mostly in the Czech language no equivalent. Therefore it was necessary to suggest for these words suitable Czech translation. In some cases, the words with zero equivalence remained untranslated in the Czech language.

It was found out that the selected text is written in scientific style and has the following features: informative function, a specialized topic, it is intended for a specific group of people and the reader and the author are required to have knowledge in the field of IT. The text uses technical terminology and contains no expressive words. The text is typical for its objectivity and condensation (participles, gerunds, hyphenated compounds) and impersonality (passive voice), the text uses connectors and phrases typical for scientific text.

In the next part was introduced the commentary containing macroanalysis and microanalysis. The content of macroanalysis was commenting on the source text from the grammatical and lexical point of view, further was described source, topic, author, reader, etc. Microanalysis dealt with particular problems, which occurred during translation.

Further a glossary of technical terms was attached with possible Czech equivalents and their meaning.

At the end of the thesis is a chapter dealing with the topic of anglicisms. A brief characterization of anglicisms was described there, their development and formal adaptation during their adopting into the Czech language. The examples related to this field were mentioned.

This work may be useful for several reasons - the terminology from the given field, process of translating of similar texts, information about used anglicisms. While working on this thesis the authoress has extended the knowledge of translating, the author has gained a general awareness about analysis procedure of purely scientific texts and has enriched the vocabulary.

# 9 Endnotes

1. Levý, J. *Umění překladu*, p. 21
2. Ibid., p. 42
3. Knittlová, D. *K teorii i praxi překladu*, p. 19
4. Knittlová, D. *Překlad a překládání*, p. 40
5. Ibid., p. 14
6. Ibid., p. 17
7. Ibid., p. 208
8. Ibid., p. 203

# 10 Bibliography

## 10.1 Print Sources

BARCLAY, K., SAVAGE, J. *Objetct-oriented design with UML and Java*. Great Britain:Butterworth-Heinemann, 2004. ISBN 0-7506-6098-8.

COLLYAH, B. a kolektiv, *Anglicko-český slovník*. Praha:FIN PUBLISHING s.r.o., 2006. ISBN 80-86002-65-9

KNITTLOVÁ, D. *K teorii i praxi překladu.* Olomouc:Univerzita Palackého v Olomouci, Filozofická fakulta, 2010. ISBN 80-244-0143-6.

KNITTLOVÁ, D., GRYGOVÁ B., ZEHNALOVÁ Jitka. *Překlad a překládání*. Olomouc:Univerzita Palackého v Olomouci, Filozofická fakulta, 2010. ISBN 978-80-244-2428-6.

KUFNEROVÁ, Z. *Překládání a čeština*. 1. vydání. Jinočany:H & H, 1994. ISBN 80-85787-14-8.

LEVÝ, J. *Umění překladu*. 4. upravené vydání. Praha:Miroslav pošta - Apostrof, 2012. ISBN 978-80-87561-15-7.

MOUNIN, G. *Teoretické problémy překladu*. Praha:Karolinum, 1999. ISBN 80-7184-733-X.

ŠIMÁČKOVÁ, V., MIŠTEROVÁ, I. *Master English Tenses*. Plzeň:Západočeská univerzita v Plzni, Filozofická fakulta, 2008. ISBN 978-80-7043-716-2.

## 10.2 Internet Sources

KOUDELKOVÁ, J. *Topzine* [online], 2010. Available from: http://www.topzine.cz/anglicismy-mozna-prevalcuji-cestinu. [Retrieved 4 April 2014].

KOVAL, F. *Banan.cz* [online]. Available from: http://www.banan.cz/serialy/Java/Java-Tridy-29-dil. [Retrieved 15 April 2014].

LEAHY, P. *About.com* [online]. Available from: http://java.about.com/od/javasyntax/a/nameconventions.htm. [Retrieved 15 April 2014].

PROŠEK, M. *Český rozhlas* [online]. Available from: http://www.rozhlas.cz/plzen/jazykovykoutek/_zprava/anglicismy--322943. [Retrieved 4 April 2014].

REJZEK, J. *Naše řeč* [online], 1993. Available from: http://nase-rec.ujc.cas.cz/archiv.php?art=7106#_ftn1. [Retrieved 8 April 2014].

*Anglistika* [online]. Available from: http://anglistika.webnode.cz/products/scientific-style/. [Retrieved 20 March].

*Free Dictionary* [online]. Available from: http://www.thefreedictionary.com/. [Retrieved 11 February 2014].

*IBM* [online], 2004. Available from: http://publib.boulder.ibm.com/infocenter/rsdvhelp/v6r0m1/index.jsp?topic=%2Fcom.ibm.rsd.nav.doc%2Ftopics%2Fcanalysismodel.html. [Retrieved 7 March 2014].

*SPOT - Slovník odborné terminologie* [online]. Available from: http://spot.zcu.cz/. [Retrieved 17 January 2014].

*Stackoverflow* [online], 2013. Available from: http://stackoverflow.com/questions/18219339/trouble-understanding-object-state-behavior-and-identity. [Retrieved 12 March 2014].

## 10.3 Electronic Source

*PC Translator 2012*. [CD-ROM]. Electronic dictionary.

# 11 Abstract

The bachelor thesis "Translation from the field of IT and anglicisms related to this field" is divided into two main parts. This is theoretical and practical part. The theoretical part is mainly the processing of basic knowledge of the theory of translation. This section deals with the process of translation, equivalence and describes the main types and procedures of translation. Further is presented the description of scientific style, in which the selected text is written. Then, this theoretical knowledge is applied in the practical part, in which is presented the translation from English into Czech. For the translation the first chapter of a book was selected. The title is "Object-oriented design with UML and Java" by K. Barclay and J. Savage. This is a book designed especially for programmers. In the practical part a commentary on translation can be found, which consists of a macroanalysis and microanalysis. Further a glossary of technical terms is presented that are contained in the selected text. At the end of the thesis is the chapter that devotes to the topic of anglicisms and examples related to this field.

# 12 Resumé

Bakalářská práce s názvem "Překlad z oblasti IT a používání anglicismů v tomto oboru" je rozdělena do dvou hlavních částí. Jedná se o teoretickou a praktickou část. Obsahem teoretické části je zpracování základních poznatků z oblasti teorie překladu. Tato část se zabývá tématem překladatelského procesu, ekvivalence a popisuje hlavní typy a postupy překladu. Dále je zpracována charakteristika odborného stylu, ve kterém je psán vybraný text. Tyto teoretické vědomosti jsou dále aplikovány v praktické části, ve které je zpracován překlad z angličtiny do češtiny. Pro překlad byla vybrána první kapitola knihy s názvem "Objektově orientovaný návrh s UML a Javou" od K. Barclay a J. Savage. Jedná se o knihu určenou především pro programátory. V praktické části se nachází také komentář k překladu, který se skládá z makroanalýzy a mikroanalýzy. Dále je vytvořen slovník odborných termínů, které jsou obsaženy ve vybraném textu. Na konci práce je kapitola, která se věnuje tématu anglicismů a jsou zde příklady vztahující se k této oblasti.

# 13 Appendix

On the following pages is the appendix containing the source text that was used for translation from English into Czech. The first chapter of a book was selected. The title is "Object-oriented design with UML and Java" by K. Barclay and J. Savage.

# 1

# Object Technology

This book is concerned with Object-Oriented Design, the Unified Modelling Language (UML) and the Java Programming Language. It seeks to demonstrate that a Java application, no matter how small, can benefit from some design during its construction. Various aspects of that design are captured and documented with the UML.

In this book we are primarily concerned with the middle ground between object-oriented design and implementation. Many textbooks exist that are solely concerned with the Java programming language (see the bibliography). These books give little or no explicit consideration to the question of design. A much smaller number of analysis and design books have been published. They often say little on the matter of realizing their designs. Here, we seek to offer a bridge between the two.

The benefit from this approach is that there is a significant shift of emphasis away from detailed programming issues on to the higher ground of analysing the meaning and accuracy of the design. Further, as mappings from the design to the implementation language are established, we recognize emerging and repeated patterns and consequently much of the programming activity often collapses into a coding chore.

This introductory chapter offers a roadmap into the remainder of the book. Here, we present the essence of object-oriented computing, and provide the necessary introductory background. We examine the fundamentals of object-oriented computing including modelling, analysis, design and implementation. The concepts are framed around everyday illustrations in which we concentrate on introducing the vocabulary of object-oriented systems.

Computer technology has developed extremely quickly since its inception. Today, computer-based systems impact on much of our lives in many spheres including banking, medical, flight reservation, educational and military applications. They are distinguished by having large amounts of software at their core. The capabilities of these systems are derived from the complex computer programs that control them.

Although we better understand the process of developing computer software, we frequently deliver it late and over budget. Often the software fails to do what the user requires and is difficult to maintain and modify. These remarks have always applied to the computer software industry, and while we have improved the technologies to support the development process, they have not fully matched the size and complexity of contemporary systems. Object technology is considered the best to deliver on these challenges, offering us the means to improve application development, reliability and scalability.

## 1.1 Background

During the 1990s object technology entered the mainstream computing landscape. The two primary fronts were in programming languages and in the introduction of object-oriented methods. The development of object-oriented methods was the subject of much research by both organizations and individuals. Notable leaders include Grady Booch (Booch 1991) and Jim Rumbaugh (Rumbaugh 1991). The various approaches promoted by these and others each had some merit but also had the effect of fragmenting the industry.

In the mid-1990s Booch and then Rumbaugh, and later Ivor Jacobson, formed the Rational organization (http://www.Rational.com). The aim was to combine their individual approaches and the contributions of others. Their efforts were offered for public scrutiny, thereby obtaining industry acceptance for a single unified object-oriented (OO) notation. It offers a means for capturing and recording the various elements of an object-oriented analysis and design (OOAD). During the late 1990s Rational published a number of versions of the UML. Subsequently in 1997 the Object Management Group (OMG) approved the UML as a vendor-neutral standard.

### *1.1.1 Modelling*

In the same manner that an architect's blueprint presents design details for a building, the UML allows software models to be constructed, viewed and manipulated during analysis and design. *Modelling* is a proven technique used in a variety of disciplines. For example, engineering models are used in the design and development of motor cars (automobile engineering), aeroplanes (aero-engineering) and bridges (civil engineering). Similarly, meteorologists have developed mathematical models to predict weather patterns.

Models are central to many human activities. They provide a blueprint for some artefact we wish to manufacture or understand. A blueprint offers a measure of repeatability ensuring standardization of the product. This is as equally important to the software customer as it is to a customer purchasing, say, household goods such as a television or a washing machine.

Through a model we aim to provide a better understanding of the system under development. Models aid our understanding of especially complex systems and help ensure we have correctly interpreted the system under development. For example, automobile engineers use models to design new motor cars that meet a number of criteria including their aerodynamics. The cars can be prototyped in fibreglass and tested in wind tunnels. A similar argument can be applied to software development whereby a model can be used to ensure all its requirements are met.

A model also permits us to evaluate our design against criteria such as safety or flexibility. Similarly, UML models permit an application's design to be evaluated and critiqued before implementation. Changes are much easier and less expensive to make when they are made in the early phases of the *software lifecycle*.

Models help us capture and record our software design decisions as we progress toward an implementation. This proves to be an important communications vehicle

between the development team members as well as between them and the customer. Development team members can discuss their designs with the client to ensure they have fully understood his needs.

Models can be layered and hence provide varying levels of detail. Abstract software models omit large amounts of fine-grained detail and permit us to gain a high-level view of the system and its architecture. These views permit us to focus on various parts of the system without recourse to the details of program code. Repeated refinement of these models can be used to progress them toward the final code.

An architect when designing a building often constructs a number of diagrams that present it from a variety of perspectives. One view is, of course, the structure of the building and is vital to the construction company. This same view is also important to the customer since it reveals the details of the accommodation, its layout and the access to stairs, elevators, etc. However, electrical (or plumbing) contractors are more interested in the run of electrical circuits (water supplies and drainage pipes) and their supply to the building from the utility companies. Hence an architect would construct a number of blueprints highlighting these various facets.

In a similar manner, the software architect can offer a range of UML diagrams that view the system from different perspectives. Some give a static view of the application with the architectural configuration of the objects as the primary focus. Other UML models emphasize the dynamic behaviour of the objects and their interactions.

### 1.1.2 UML

The UML defines a diagrammatic notation for describing the artefacts of an OOAD. Through the UML we can visualize, specify, construct and document our software application. As our software systems become ever larger and ever more complex we need to manage that complexity and, in a sense, simplify it so we have a better understanding of it. Often, visualizing the software graphically is more appropriate than struggling to understand it in program code.

By inspecting our models we can identify deficiencies in our designs as well as opportunities to enhance them. The UML acts as a specification language in which we can precisely and unambiguously capture our design decisions.

Finally, from our UML diagrams we can derive programming language code. This is referred to as *forward engineering* — the generation of code from UML models. This is an approach we advocate through this textbook. The models are at the core of our designs. The code is an outcome of that modelling activity and is itself a design document. The models dictate the code that we ultimately produce.

### 1.1.3 Analysis and design models

An *analysis model* used in software development aims to document various facets of the real world problem that we are modelling. In an object-oriented system development this would typically involve identifying the significant application objects and the application processing to be performed.

Development methods that pre-dated object technology often deliberately delineated between the analysis phase and the design stage. Commonly, they would also use a linear or *waterfall model* for the development process in which the design stage only follows after all the analysis has been completed. Further, these separate analysis and design stages often resulted in a conflict between them, especially where different models and notations were deployed.

Object-oriented methods are characterized by using the same modelling concepts throughout the software lifecycle. This way, the solution that emerges during analysis is carried through into design and finally to code. Objects identified during analysis should also be present in the final code. This offers a seamless integration of the stages, not otherwise found with other approaches. The design models augment the analysis models with additional detail and, perhaps, introduce further low-level system objects required in the implementation.

### *1.1.4 Development process*

The UML is a modelling language. It has no notion of a development process, which must accompany a method. The dictionary defines a method as a systematic or orderly procedure. The authors of the UML understood this distinction and deliberately sought to separate the language used to document a software design from the process used to develop it. They recognized that processes are influenced by many considerations such as the nature of projects and the culture of organizations.

Object-oriented design is usually conducted within an *iterative process*. This is vital to ensure that we can revisit earlier decisions when corrections or modifications are necessary. This is not unreasonable. After all, initial design decisions may require revision, especially in new projects or in those that are less well understood by the development team. The iterative process continues until the full system is developed.

In common with many object-oriented developers, an iterative process is also accompanied by an *incremental* style of development. Each increment introduces some additional functionality on to the previous stage. Often, the new increment only adds a small feature so that we can fully test it and its effect on the existing system and its architecture.

Each iteration needs to be accompanied by an objective that can be checked. Otherwise, there is a danger of the process degenerating into *undisciplined hacking*. An iterative approach is further enhanced when the customer is closely involved in the system development. Each new iteration can be presented to the customer to obtain feedback and to ensure his active participation throughout the project.

## 1.2 Using the UML

The next chapter presents a detailed discussion of the UML. Here, we simply consolidate our earlier discussions by considering the more important elements of an object-oriented software system. This will act as an introduction to the more detailed discussions that follow.

### 1.2.1 Objects: combined services and data

An object-oriented system comprises a number of software objects that interact to achieve the system objective. The software objects usually mimic the real-world objects of the application domain. The real-world objects may have a physical presence or may represent some well-understood conceptual entity in the application. For example, in a university application we might have software objects that represent students. Equally, we may have software objects representing programmes of study at a university even though they have no physical existence.

Objects are characterized by having both *state* and *behaviour*. The state of an object is the information an object has about itself. For example, a student object may have a name, a date of birth and a university matriculation number. Equally, a programme of study object might have the name of the programme, its duration and the name of the programme leader. The behaviour of an object describes the actions the object is prepared to engage in. For example, we might ask a programme of study object for its duration. We might ask a student object for its age. This would involve the student object performing a calculation based on its date of birth and today's date.

The behaviour of an object is described by the set of *operations* it is prepared to perform. One object interacts with another by asking it to perform one of its advertised operations. This interaction is achieved by one object sending a *message* to another. The first object is known as the *sender object* and the second object is known as the receiver or *recipient object*. The only messages an object can receive belong to the set of operations it can accept. In the UML, objects and message passing are usually captured by a *sequence diagram* as shown in figure 1.1. Here a university object is shown sending the message getAge to a student object.

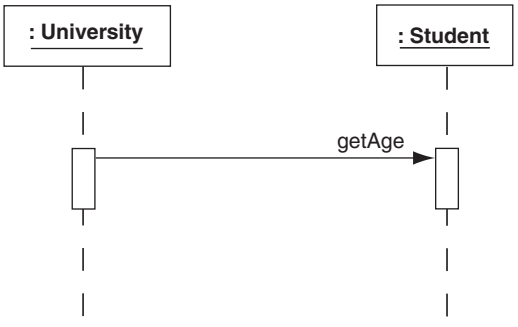



**Figure 1.1** *Message passing in a sequence diagram*

When an object receives a message it performs some action. This action is described by a *method*. A method is the processes the receiving object follows when servicing the message. For example, if a university object sends a message to a student object asking for its name, then the student object simply replies to the sender with one part of its state, namely, the name. However, if a university object sends a student object the message asking for its age, then the method that student object must follow is more elaborate. First, it must obtain today's date. This might be achieved by the student object

sending a message to some calendar object. The student object then has to perform some complex date arithmetic on its date of birth and today's date to determine its age. In the sequence diagram of figure 1.1 this processing is shown as an *activation*, the rectangle adjacent to the message arrow.

Figure 1.1 also implies *message propagation*. When one object receives a message it often sends a cascade of other messages to other objects. The university object sends a message to the student object asking for its age. In turn, it sends a message to some calendar object requesting today's date. This example also demonstrates that an OO system is a mix of objects interacting to achieve the required objective.

A university would typically have a large number of students. Unlike real students, all student objects exhibit the same behaviour and carry the same knowledge about themselves. We might model a student object with a name, date of birth and matriculation number. The actual state values for two student objects are presumably different since university matriculation numbers are unique. With a large university population we might, however, expect two or more students with the same name or two or more with the same date of birth.

They are, however, all subject to the same behaviours. If one student can be asked for their age by sending some suitable message, then all students can be sent this message. How is this determined? All of our student objects support a single abstraction that we may choose to call Student. Other abstractions from this problem domain might include University, ProgrammeOfStudy and Tutor. We refer to the abstraction as the *class* of the object.

A class is effectively a blueprint or template that fully describes the abstraction. The Student class describes any number of student objects. The Tutor class describes any number of tutor objects. The class describes the information an object holds to represent its state. The items of information are called *attributes* (sometimes also called *properties*). The class also defines the behaviours of such objects, listing the operations they can perform, i.e. the messages they can receive. The effect of these operations is described by its method. Figure 1.2 shows a simplified *class diagram* for a Student class.

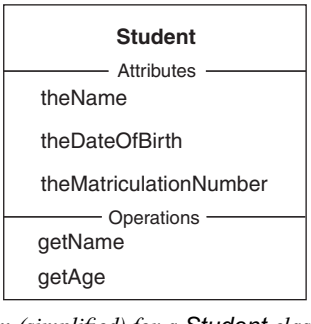



**Figure 1.2** *UML class diagram (simplified) for a* Student *class*

In this figure we have a class Student with two operations and three attributes. Any student object we create from this template will have a state comprising three values for the attributes, i.e. theName, theDateOfBirth and theMatriculationNumber. Further, any

Student object can be sent messages to obtain their name or their age, i.e. getName and getAge.

Figure 1.3 shows how in the UML we present a particular example of an object from some named class. This we refer to as an *object instance* or simply an object. The upper part of the figure names the class to which the object instance belongs, here Student. It also labels the object with some identifier by which we can refer to that object (s1). The lower part presents the attribute values maintained by the instance and represents its state. Here, for example, this particular Student instance has theName attribute with the value Ken Barclay as part of its state. Such a diagram element may be part of a much larger *object diagram* (or *collaboration diagram*) that we describe with the UML (see chapter 2).

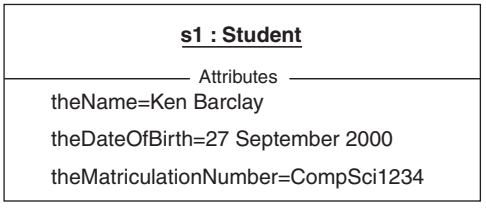



**Figure 1.3** *An object instance*

As a further example, consider how we might model a bank account. An account can exhibit a variety of behaviours such as debiting or crediting some monetary amount, or requesting the account's current balance. These behaviours give rise to some of the likely account operations. Debit and credit transactions document the amount involved in the transaction, changing the balance for an account. The balance, along with the account number must be maintained by each bank account instance. Every example of a bank account carries its own data values for these attributes (see figure 1.4).

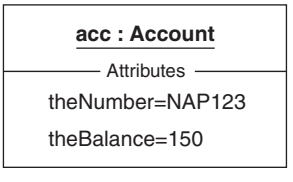



**Figure 1.4** *A bank account instance*

To model an account as an object we describe its behaviours as operations and its state with attributes. During the execution of a system, an account object is requested to carry out its various operations, changing its attribute values as needed to reflect the effect of its actions. For example, in figure 1.4, a debit operation applied to such an account object results in a change to the value of theBalance attribute.

Some operations are used to get information about an object, while others have some effect on an object's state. The operations that only give information about an object are referred to as *enquiry operations*. They enquire about some state information held by the object. The operation to obtain the value of an account's balance is of this type.

The operation that performs a debit transaction on a bank account object changes the current balance it holds. This category of operation is described as a *transformer operation*. A transformer operation changes one or more of the object instance attribute values. Both operations refer to the values of the object's attributes, collectively the *state* of the object. Transformer operations result in a state change, while enquiry operations do not usually affect the state of the instance. In the class diagram of figure 1.5 we recognize the operation getBalance as an enquiry operation, while debit and credit are transformer operations.

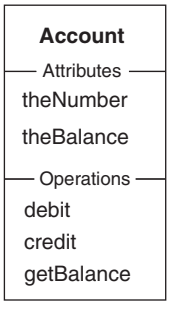



**Figure 1.5** *Class diagram with* Account *class*

### 1.2.2 Objects make excellent software modules

The concept of an object is both simple and yet extremely powerful. Objects make ideal software modules. Each object instance forms a self-contained entity. Everything an object knows is expressed in terms of its attributes and everything it can perform is expressed by its list of operations. For this reason, objects are described as *highly cohesive*. All the characteristics of an object provide some well-bounded behaviour for the particular abstraction they represent (*encapsulation*).

Consider a motor car. A car has various controls that are used to control and operate it. The gear shift, for example, is used to change gear on a manually operated vehicle or to select the drive on one that is an automatic. Cars usually have tachometers to show the speed of the vehicle. Other controls include the accelerator and the brake.

The internal components of the car are implemented by the many mechanical and electronic devices contained within the body of the car (usually under the engine bonnet). This metal carcass isolates the driver from the internals. Since the driver has no direct contact with these components there is no likelihood that he will damage himself or the car. Consider the position where, instead of an accelerator pedal, the driver controls the speed by using a screwdriver on some internal control screw. The accelerator would usually have some restricted amount of movement, limiting the driver to a certain maximum speed. Without this restriction it is possible that adjustments made directly to the control screw may set the speed above its maximum safe working level and consequently damage vital components.

Equally, by isolating the driver from the internals, and only permitting a trained engineer access to the components, a second benefit is obtained. If the car develops a fault, then an engineer may replace the faulty component. The change has, however, been effected without any operational change. The driver still uses the car in exactly the same way. Only the repair of the faulty component has changed.

These ideas are also mirrored in object technology. They present the same benefits to software as they do to cars. In a software object the attributes are hidden from a user but may be replaced without affecting user software that relies on that object abstraction (*information hiding*). In a software object behaviour is defined by its operations and not by its private representation. The operations govern what we can ask a software object to perform. If we supply an unacceptable value when invoking an operation, then the effect can be denied, ensuring the correct state of the object. For example, a debit operation applied to a bank account might only be permitted if sufficient funds are available.

Since object attributes and operations only define their abstraction and no other, they usually exhibit *loose coupling* with other objects. This is highly desirable because *strong coupling* makes software components harder to understand, change or correct. For example, when defining an object's operation we need not concern ourselves with the requester (client) of that operation. Only the effect of that operation on the receiving object's attributes need be considered. Equally, the client requesting a service need not be concerned with how that request is achieved by the recipient object.

### 1.2.3 Object interaction is expressed as messages

Figure 1.1 demonstrated that objects interact with each other by sending a message from the sender requesting the recipient object to carry out one of its advertised operations. In figure 1.1 a University object requests the age of a Student object. Equally, a bank object may send one of its account objects a credit operation. Here, the bank is the sending object and the account is the receiving object. A message identifies the recipient object and the name for the operation to be performed. The message name represents one of the operations of the class to which the recipient belongs. If the message requires any further details they are given as the *message parameters*.

To request a bank account object to engage in a transaction to debit it by some monetary amount, some sending object, such as an automated teller machine (ATM), might send the account object the message:

```
acc debit 50
```

Here, acc is the identifier of the receiver object which is some bank account object, debit is the operation it is being asked to execute, and 50 is the actual parameter informing the receiver account of the amount involved in the transaction. The UML *collaboration diagram* in figure 1.6 portrays this message passing between objects. In the diagram we have two objects: an Account object with identifier acc and an anonymous (no identifier) ATM object. The latter sends a message to the Account object to perform the operation debit with the actual parameter value of 50.

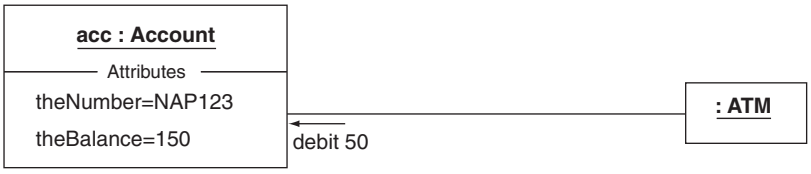



**Figure 1.6** *Message from an ATM to an account object*

When a message is received by some recipient object then an action is performed. This action usually involves some or all of the values of the attributes representing the state of the receiving object. The action will also use any message parameters. The logic associated with this action is described by the method for the operation. The *method* refers to the *algorithm* that is applied when an operation is executed. The method for the debit operation applied to an Account object involves reducing the value of theBalance attribute by the actual message parameter value of 50.

We have noted how transformer operations usually result in a state change to the receiving object, while enquiry operations merely request information from it. The only means of communication is a message sent from a sender to a receiver. In the case of an enquiry operation a secondary information flow is observed. Here, the sender is expecting a response from the receiver in the form of a *return value*. Occasionally, transformer operations also supply return values, say, to report to the sender that the designated task has been completed successfully. Be sure to recognize that a return value is not another message.

Observe also the asymmetry of the messaging concept. The recipient object, when defining its operation's logic, does not concern itself with the object that is sending the message. Equally, the sender need not be concerned with how the operation is implemented by the recipient. As noted in the preceding section this greatly assists with the production of high quality software systems by the separation of these two concerns.

## 1.3 Classes: sets of similar objects

An object-oriented system is characterized as a set of interacting objects. It is therefore common to have more than one object of any given kind. For example, a bank will certainly have a number of customer accounts each of which carries out the same actions and maintains the same kind of information. The single class Account (such as figure 1.5) could represent the entire collection of account objects (such as in figure 1.4). The class contains the specification and definition of its operations (methods) and its attributes. The actual accounts are represented by instances of this class, each with its own unique identifier (say, acc1, acc2, …). Each instance contains data that represents its own particular state. When an account receives a message to carry out one of its operations, it uses the method definition for the operation given in its class and applies it to its own attribute values.

Figure 1.7 shows the Account class and two instances of that class. The instances have identifiers acc1 and acc2. The Account class has three services provided by the operations debit, credit and getBalance. The attributes maintained by every

instance of this class have their own values for theNumber and theBalance. For example, in the instance with identifier acc1 these attributes are respectively DEF456 and 1200. The message:

```
acc1 debit 50
```

results in the execution of the method for the debit operation applied to the Account object acc1. This operation might be defined in terms of subtracting the value of the message parameter 50 from the value of the attribute theBalance presently held by the Account object acc1. The effect of this transformer operation produces a state change in the object acc1, reducing theBalance to 1150.

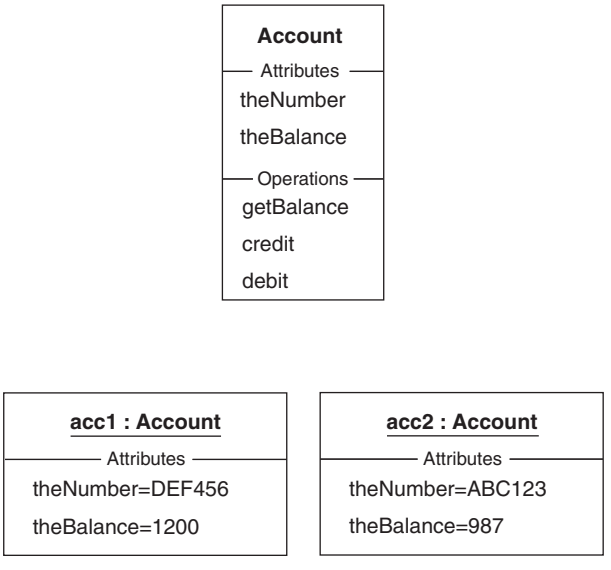



**Figure 1.7** *The Account class and two instances*

Normally objects do not exist in isolation. They form relationships with each other and related objects engage in message passing. In the object diagram of figure 1.8 we show two Account objects that are related to the same Bank object. The Account objects are not related to each other. Thus the Bank object can send messages to either or both Account objects and the latter can send messages to the Bank object. Significantly, since there is no relation between the Account objects therefore they cannot engage in message passing.

The classes and the relationships between them are modelled in the class diagram in figure 1.9. The annotation 0..* indicates that one Bank object can be related to none (0) or more (*) Account objects. We shall have much more to say on these diagrams in the next chapter. The object diagram of figure 1.8 is a particular example of a configuration of objects described by this class diagram. In principle, there are an arbitrary number of object diagrams that are based on a single class diagram. Thus a class diagram is a abstract description of all possible configurations of objects.

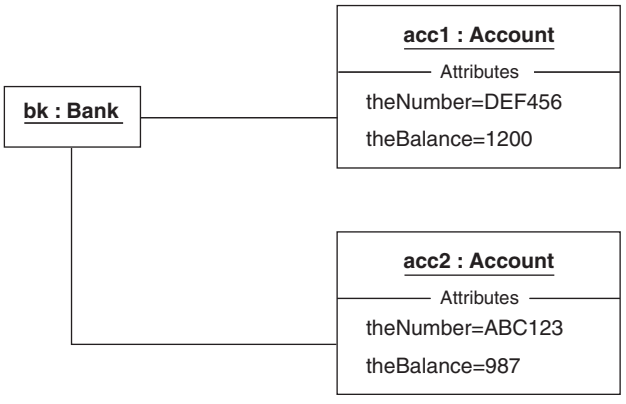



**Figure 1.8** *Objects and relationships*


**Figure 1.9** *Class diagram with a relationship*

### *1.3.1 Specialization*

Object-oriented models can give rise to many classes of objects. Such complex models may be rationalized and simplified by arranging the classes into hierarchies. The hierarchies assist in the categorization of the types of object instances represented by the many classes. This categorization of knowledge is found in many science and engineering disciplines and greatly assists in simplifying complex systems.

Many people, whatever their particular kind, share common characteristics. These characteristics include both state information and behaviours. For example, name, home address and social security number are data shared by all people. Equally, all share the same behaviour of being able to ask a person for their age. People may be further specialized as a particular kind of person, e.g. student. In addition to all the qualities associated with people they also have additional characteristics peculiar to being a student, such as a matriculation number. Everything that applies to an ordinary person also applies to students and every instance of a student is implicitly an instance of a person.

The Student class is described as a *specialization* of the Person class. Conversely, the class Person is a *generalization* of the class Student. The specialized class Student is said to *inherit* all the features of its generalization class Person. Thus if any person has a name, then so do students. If we can ask a person for their age then we can do the same with a student. In fact, any operation that may be applied to a Person instance may also be applied to a Student instance. The converse, however, is not true. A Student object may have attributes and operations peculiar to it, such as a matriculation number. Since a Person is a generalization of a Student, only those common characteristics are applicable to Persons. Hence we are not permitted

to ask a person for their matriculation number since this is only applicable to students.

Specialization is the mechanism by which one class is defined as a special case of another class. The specialized class includes all the operations and attributes of the general class. The specialized class is said to inherit all the features (or characteristics) of the general class. The specialized class may introduce additional operations and attributes peculiar to it. In addition to the operations inherited, the specialized class may choose to *redefine* the behaviour of any one of these. This, as we shall see shortly, is used when the specialized class has a more specific way of defining that behaviour. The specialized class is commonly known as the *subclass* and the general class its *superclass*.

Specialization is often described as *programming by difference*. Since the Student class inherits from the class Person, then the Student class need only implement those differences between itself and the generalized Person class. So, for example, the Student class need only introduce the additional attributes peculiar to students, say, a matriculation number. Further, through inheritance, the specialized Student class need not reprogram the inherited operations. The difference is any additional operations and any redefined operations. The subclass then benefits from a significant amount of *code reuse*.

In our banking illustration, the class Account could be specialized into two subclasses CurrentAccount and DepositAccount. The specialization relation is illustrated as a directed arrow from the subclass to the superclass as shown in figure 1.10. Each inherits the general characteristics of its common superclass. Either subclass may then add to the set of operations and attributes of the superclass, or *redefine* the behaviour of one or more inherited operations.

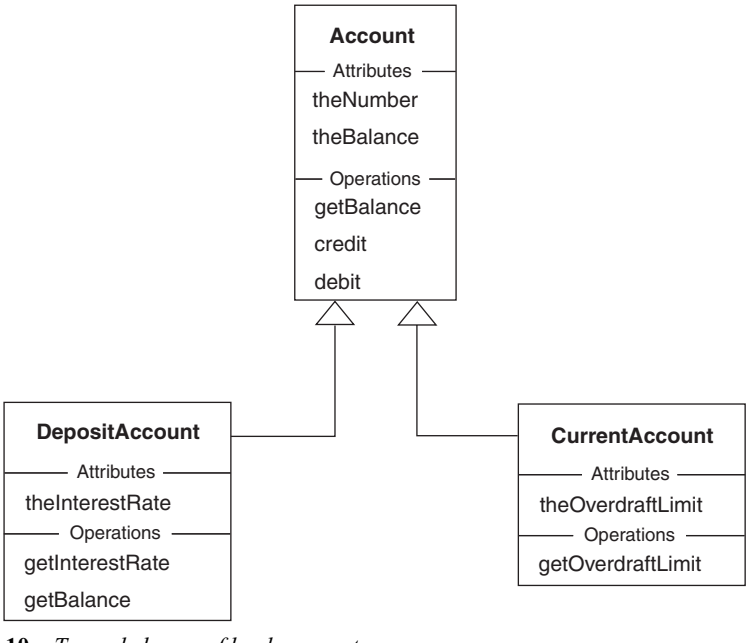



**Figure 1.10** *Two subclasses of bank account*

In Figure 1.10 any instance of the class CurrentAccount has the attributes theNumber, theBalance and theOverdraftLimit. The latter attribute is introduced in the CurrentAccount class itself, while the other attributes are inherited from the Account superclass. Equally, any instance of CurrentAccount can respond to the messages debit, credit, getBalance and getOverdraftLimit. Again, the first three operations are inherited from the superclass and the last is defined for the CurrentAccount class. A similar arrangement applies to the DepositAccount class in which the attributes of an instance are theNumber, theBalance (both inherited) and theInterestRate (declared by the class). The operations are debit, credit (both inherited), getInterestRate (defined by the class) and getBalance (redefined).

Redefining the definition of an operation in a subclass permits a specialized implementation of the method. For example, the operation getBalance in the Account class may be defined to simply return the present value of the attribute theBalance. In the subclass DepositAccount the reappearance of the operation getBalance indicates a redefinition that might, for example, deal with any interest accruing.

### *1.3.2 Polymorphism*

Through inheritance, one class is formed as a specialization of an existing class, inheriting all the features of that existing class. This proves to be a particularly important concept that supports re-usability of existing code. Inheritance also gives rise to the notions of *polymorphism* and *dynamic binding*. These additional concepts provide support by which software systems may be modified to accommodate changes to its specification.

The dictionary definition for polymorphism is "having many forms". The class definition for DepositAccount declares the explicit specialization from class Account and reveals that a DepositAccount is an Account with additional attributes, operations and redefined operations. Hence an instance of DepositAccount may be substituted for an Account instance. This is permitted since an instance of the class DepositAccount can be sent the same messages as an instance of the Account class. Equally, an instance of CurrentAccount may also be used where an Account instance is expected. This means, for example, a Bank object may be introduced with a number of Account objects associated with it. The Bank does not need to concern itself with whether they are CurrentAccount objects or DepositAccount objects. They are all some kind of Account.

This approach is radically different from that which is employed in conventional systems where it is necessary to populate code with complex selection statements to identify the kind of account then execute some appropriate logic. In these systems the determination of the account kind lies wholly with the programmer. In object-oriented systems responsibility for this selection is given to the programming environment.

Figure 1.11 presents a class diagram in which a number of accounts are held or maintained by a bank. A particular instance of the class Bank is responsible for zero or more instances of the various kinds of bank accounts, some of which are DepositAccounts and some CurrentAccounts. The labelled line is an *association* demonstrating a

one-to-many relationship between the Bank class and the general Account class. This kind of object model is the subject of this book and is formally introduced in the next chapter.

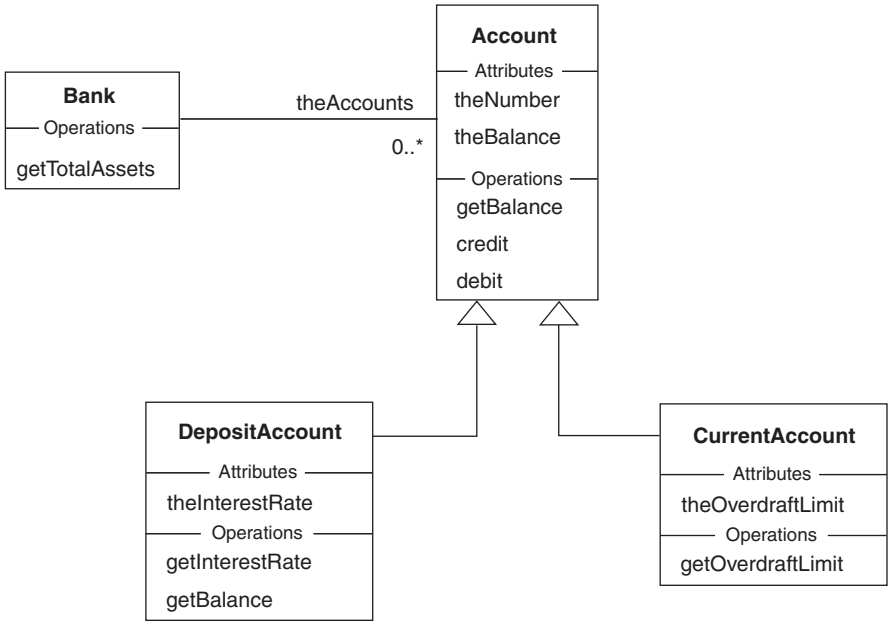



**Figure 1.11** *Bank/Account class diagram*

When a Bank object sends the message getBalance to each of the many Account objects it holds, then the definition for this operation in the Account class is used. Some accounts will, of course, be interest-bearing DepositAccounts that, through redefinition of the getBalance method, have a different way of computing the amount of interest. To obtain the correct selection of method we defer the choice of method to execute until *run-time* using a mechanism known as *dynamic binding*. This is done by recording that the *polymorphic effect* is required on the getBalance operation. If the operation getBalance in the class Account is polymorphic, then when the message is sent to each account instance the appropriate definition of the operation is executed according to the class of the receiving object. Effectively, the receiving object knows to which class it belongs and executes the appropriate method.

Thus when the getBalance message is received by a DepositAccount instance, then the redefined version of the method from that class is executed. When a CurrentAccount object receives the same message, then since that class does not redefine the operation, the method executed is that defined and inherited from the superclass Account. The UML collaboration diagram in figure 1.12 demonstrates how the Bank object determines the value of its total assets. Each account is either an instance of a DepositAccount or an instance of a CurrentAccount. The Bank is unaware of this fact and simply sends the message getBalance to each. When the single CurrentAccount object receives the message getBalance it simply executes the method inherited from

the Account superclass. The two DepositAccount objects use the redefined method in their subclass.

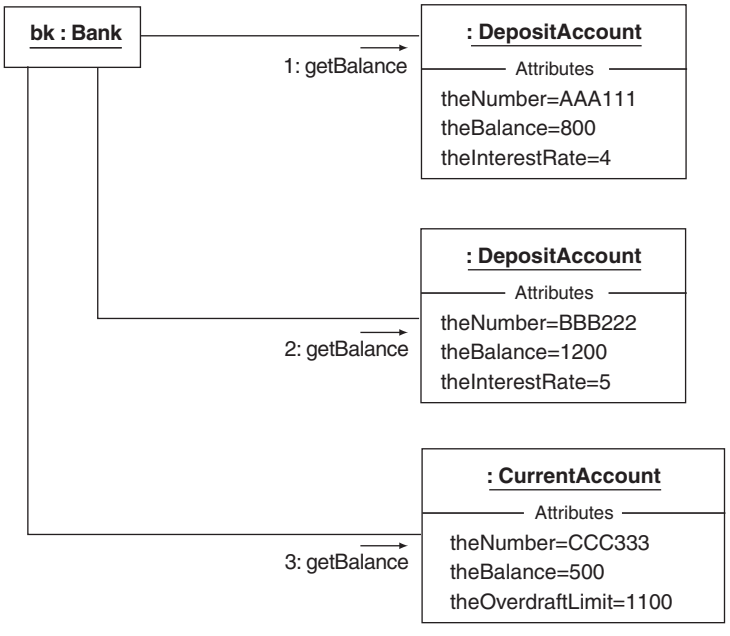



**Figure 1.12** *The dynamic binding effect*

Polymorphism contributes to program maintenance. If we were to add a new kind of account to the system, say the class SavingAccount as a specialization of the Account class, it might implement its own getBalance operation. The Bank class does not need to know about this enhancement. Again, it sends the getBalance message to every account. If a particular instance happens to be from this new class it will choose its own definition for this operation.

## 1.4 Tools

Today, modern software development normally takes place with the support of software development tools. They are generally described as *Computer Aided Software Engineering* (CASE) tools. Many commercial CASE tools are large and complex software systems that support many of the stages of the software development process. They are often known as upper-case tools because of their support for most aspects of a process. By comparison, lower-case tools provide less support but usually require a much shorter learning curve.

To support the reader throughout the remainder of this textbook, the authors have made available a lower-case tool ROME (see appendices A and B). That way, the reader can follow the book's content and have access to CASE tool support for various parts of the discussion. Whereas upper-case tools strive to automate activities, ensure consistency

across the models, provide various management reporting, etc., ROME does not include these features to ensure we have a much simpler tool to operate. At the end of this study however, we expect that the reader should be capable of progressing to the more advanced commercial tools. For example, consult the website http://www.rational.com, http://www.togethersoft.com.

The ROME modelling tool has been used throughout this book to create the many UML diagrams shown. The current version of ROME supports the majority of diagrams described by the UML. Further, the class diagrammer in ROME is used to generate the Java code.

See the section entitled "Software distribution" in the Preface for details of how to obtain and install the supplied software.

## 1.5 Summary

1. The Unified Modelling Language, UML, is an internationally agreed notation for recording the various elements of an object-oriented analysis and design. The UML defines a number of views of a system through various diagrams such as class and collaboration diagrams.
2. The UML must be augmented with a process to guide the development of the software. In an OOAD the same modelling concepts are used throughout the software development process.
3. An object-oriented system is characterized as a set of communicating objects.
4. An object is a set of operations together with a state that the object retains between invocations of any of its operations. Transformer operations result in a change to that state while enquiry operations report on the state.
5. An object instance is a particular example of an object from some named class and can be shown in a UML object diagram. A class is a blueprint or template describing an arbitrary number of such instances and is presented in a class diagram.
6. Objects interact through message passing shown in either UML collaboration or sequence diagrams. One object sends another object a message that invokes one of the recipient's operations. The message is bound to the operation's definition given in its class or its superclass.
7. Classes may be classified into a hierarchy starting from the general and leading to the more specific. A subclass is a specialization of its immediate superclass. A subclass inherits all the features of its superclass. It may add further features and redefined operations. An instance of a subclass is an instance of its superclass and may substitute at any time for an instance of the latter.
8. Inheritance also gives rise to the notions of polymorphism and dynamic binding. A dynamically bound, polymorphic message sent to an object binds to the operation definition in the class to which the object belongs.

## 1.6 Exercises

1. Explain why modelling is central to many human activities. Other than those already described, give one further example for the use of models.