ETH Institut für Verhaltenswissenschaft, Departement Informatik

Leitprogramm

Einführung in die Objekt-Orientierte Programmierung

Schultyp: Gymnasium oder Hochschule

Voraussetzungen der Adressaten: Keine Programmierkenntnisse erforderlich

Bearbeitungsdauer: 12 Stunden

Autoren: Till G. Bay und Michela Pedroni

Betreuer: Prof. Dr. Ulrik Schroeder, Prof. Dr. Juraj Hromkovic

Fassung vom: 26. September 2005

Schulerprobung am: Noch nicht erprobt

Einführung

Beim Programmieren geht es darum mit dem Computer Aufgaben zu bewältigen. Dabei nutzt man die Verlässlichkeit und die Geschwindigkeit von Computern, die zum Beispiel schneller rechnen können als wir Menschen. Damit man den Computer dazu verwenden kann, muss ein Problem in einer für den Computer verständlichen Sprache formuliert werden: in einer sogenannten Programmiersprache. Es existieren viele Programmiersprachen, die sich in verschiedene Familien von Programmiersprachen einteilen lassen. Die Familie der objekt-orientierten Programmiersprachen ermöglicht es, ein Problem auf natürliche Art in einem Programm zu beschreiben. Deshalb hat die objekt-orientierte Programmierung in der Informatik einen grossen Stellenwert.

In diesem Leitprogramm lernst du objekt-orientiertes Programmieren mit der Programmiersprache Eiffel; das hier Erlernte ist jedoch auf andere Programmiersprachen anwendbar. Nach Bearbeitung solltest du die folgenden drei Lernziele erreicht haben:

- Du kennst die objekt-orientierte Denkweise und die Grundbegriffe dieser Art zu Programmieren.
- Du kannst dich in Programme, die von jemand anderem geschrieben wurden, einarbeiten und sie verändern.
- Du kannst die Programme, die zur Entwicklung von anderen Programmen benutzt werden, bedienen.

Es ist hingegen nicht ein Ziel dieses Leitprogramms, dass du ein Programm von Grund auf neu entwickeln kannst.

Heute programmiert man überall. Wir programmieren unseren Wecker, wir programmieren unser Mobiltelefon so, dass es auch Bild-Nachrichten versenden kann, und wir programmieren Webseiten für unseren Verein. In jedem Fall steckt eine Form einer Programmiersprache dahinter, deren Befehlssatz wir verwenden, um einem Gerät Anweisungen zu geben. Programmieren ist eine Kombination von Kreativität, schöpferischem Arbeiten und mathematisch-analytischem Denken. Wir finden diese gegensätzlichen Eigenschaften sehr spannend und laden dich ein, diese Erfahrung mit Hilfe des Leitprogramms selbst zu machen.

In diesem Leitprogramm lernst du die Grundkonzepte der objekt-orientierten Programmierung kennen und wendest sie beim Programmieren eines Memory-Spiels an. Zu Beginn erhältst du ein minimales Grundgerüst des Spiels, das bereits funktioniert. Schrittweise wirst du dieses Programm ausbauen, bis du ein tolles Game programmiert hast, mit dem du bei denen Eltern, Verwandten und Freunden angeben kannst. Auf geht's…

Inhaltsverzeichnis

Arbeitsanleitung

Weil du in diesem Leitprogramm von Beginn an programmieren wirst, bearbeitest du es am besten im Computerraum. Das Leitprogramm ist so aufgebaut, dass du jeweils dort weiterarbeiten kannst wo du das letzte mal aufgehört hast. Deshalb ist es wichtig, dass du dir merkst, wo du dein Projekt abspeicherst, damit du es das nächste Mal wieder findest. Es ist möglich das Leitprogramm zu zweit an einem Computer zu bearbeiten, sollte euer Computerraum zu wenig Arbeitsplätze haben.

Icons die in diesem Leitprogramm verwendet werden

kein Icon — Normaler Lehrtext hat kein Icon.

Eine Vorgehensweise wird erklärt.

Dieses Icon markiert Stellen, wo dein Wissen überprüft wird.

Hier wird Wissenssicherung betrieben - das heisst Erlerntes wird in anderer Form noch einmal erklärt, weil es so besser hängen bleibt. Diese Teile des Leitprogramms bearbeitest du für dich alleine; niemand kontrolliert dich, und du kannst sie auch mehrmals wiederholen.

Das ist eine Auflistung von Lernzielen. Hier wird beschrieben, was du nach Bearbeitung eines Kapitels gelernt haben wirst. In der Einführung auf Seite 3 findest du sogar die Lernziele für das ganze Leitprogramm.

Kapitel 1

Aufwärmen

Till G. Bay

Übersicht

Was lernst du hier?

Du lernst wie ein Programm entsteht. Anhand des Memory-Spiels, das du in diesem Leitprogramm bearbeiten wirst, programmierst du deine ersten Zeilen. Dazu brauchst du keine Programmierkenntnisse und du musst auch nur grundlegende Computerkenntnisse haben.

Was tust du?

Du liest zuerst den Einführungstext über das Programmieren und über Entwicklungsumgebungen. Danach wirst du verstehen, wie es von einem in einer Programmiersprache geschriebenem Programm zu einem durch den Computer ausführbaren Programm kommt. Dieses Wissen wendest du dann direkt am Beispiel des Memory-Programms an. Die Dateien des Memory-Programms werden als Programmierprojekt gemeinsam verwaltet. Es gibt eine Projekt-Datei, die beschreibt welche Dateien alle dazugehören. Du wirst das Memory-Projekt in einer Entwicklungsumgebung öffnen, daraus ein ausführbares Programm erzeugen und dieses Programm ausprobieren. Du wirst das Programm auch verändern, erneut eine ausführbare Version erstellen und dann die Veränderungen beobachten. Nachdem du weiter unten die Lernziele gelesen hast, kannst du mit Abschnitt 1.1 beginnen.

Lernziele

Nachdem du dieses Kapitel durchgearbeitet hast,

- kannst du erklären, wie es vom Programmtext zum ausführbaren Programm kommt
- kannst du ein Programmierprojekt öffnen und daraus ein ausführbares Programm erstellen
- kannst du erklären, welche Schritte notwendig sind, damit deine Veränderungen am Programmtext im Programm wirksam werden

1.1 Quelltext

Programme werden in einer speziellen Sprache — einer Programmiersprache — geschrieben. Diese Sprache ist viel einfacher als die Sprache, in der sich Menschen unterhalten. Sie ist auf wenige Sprachkonstrukte reduziert, in denen man einem Computer Anweisungen geben kann, was in einem Programm auszuführen ist. Die Programmiersprache hat ein sehr eingeschränktes Vokabular von Befehlen. Mit diesen Befehlen beschreibt man im Programmtext was geschehen soll. Das so geschriebene Programm wird von oben nach unten ausgeführt.

Programme, die du auf deinem Computer startest, sehen aber nicht aus wie Textdateien, die Programmtext enthalten - wie kommt das? Der Grund dafür ist ganz einfach: Es wäre sehr unpraktisch Programme als Programmtexte zu verteilen, denn es gibt sehr viele verschiedene Programmiersprachen und der Computer müsste sie alle kennen, um diese Programmtexte direkt ausführen zu können. Deshalb werden die Programmtexte zuerst noch in eine andere Sprache übersetzt, bevor man sie als Programme auf einem Computer ausführen kann. Man nennt diese Sprache sinngemäss *Maschinensprache*.

Diesen Übersetzungsschritt nennt man im Fachjargon *Kompilierung*. Das Wort Kompilieren stammt aus dem lateinischen und bedeutet zusammenstellen. Man stellt also aus dem Programmtext ein Programm zusammen. Dem Programm liegt also der Programmtext zu Grunde - man nennt ihn deshalb auch *Quelltext* oder in Englisch *sourcecode*, weil er die Quelle, den Ursprung des Programms darstellt. Wir werden fortan das Wort Quelltext verwenden.

Der Quelltext besteht aus vordefinierten Schlüsselworten. Die Schüsselworte von Eiffel die wir verwenden werden sind hier aufgelistet. Eiffel kennt noch weitere, diese verwenden wir aber nicht und lassen sie deshalb vorerst weg.

- **class**
- **feature**
- **indexing**
- **inherit**
- **is**
- **require**
- **local**
- **do**
- **ensure**
- **if**
- **then**
- **else**
- **undefine**

Abbildung 1.1: Kompilierung

- **create**
- **invariant**
- **end**

Die Kompilierung ist eher kompliziert, und wir verwenden dazu ein Programm, dass sie für uns erledigt - dieses Programm heisst *Kompiler*. Es gibt deshalb also für jede Programmiersprache einen Kompiler, der Quelltext, welcher in dieser Sprache geschrieben wurde nach Maschinensprache in ein ausführbares Programm übersetzen kann. Dieser Ablauf ist in der Graphik 1.1 dargestellt.

Ein Benutzer eines Programms, verfügt nur über das ausführbare Programm in Maschinensprache. Programmierer hantieren mit Quelltext und dem ausführbaren Programm. Wenn der Programmierer also im Quelltext etwas am Verhalten des Programms ändert, muss er den Quelltext neu kompilieren, um die Änderung im Programm zu sehen. Genau das wirst du gleich selbst tun. Zuerst musst du aber erstmal aus einer Sammlung von Quelltexten, die wir Dir geben ein Programm erstellen - dazu verwendest du eine Entwicklungsumgebung. . .

1.2 Entwicklungsumgebung

In diesem Leitprogramm erlernst du die Grundlagen der objekt-orientierten Programmierung anhand der Sprache Eiffel. Eiffel ist eine von vielen objekt-orientierten Programmiersprachen - andere, von denen du vielleicht bereits den Namen gehört hast, sind Java oder C++. Die Sprache Eiffel hat viele Eigenschaften, die sie besonders geeignet machen, um in die objekt-orientierte Programmierung einzusteigen. Doch mehr dazu später, jetzt wollen wir erst einmal ein Eiffel-Projekt öffnen.

Starte nun die Entwicklungsumgebung für Eiffel - EiffelStudio. Wenn du nicht weisst, wie man EiffelStudio startet, kannst du auf dem Zusatzblatt „Aufwärmen“ nachschauen. Das Zusatzblatt befindet sich in der Nachschlagebibliothek im Ordner.

Eine Entwicklungsumgebung ist ein Programm, mit dem man andere Programme schreiben und kompilieren kann. Eine Entwicklungsumgebung enthält also einen Editor mit dem man Quelltext schreibt und einen Button mit dem man die Kompilierung startet, sowie einen Button, mit dem man das kompilierte Programm startet. Das ist natürlich eine sehr vereinfachte Beschreibung, aber diese drei Elemente kommen bei allen Entwicklungsumgebungen in der einen oder anderen Form vor.

Normalerweise entstehen beim Programmieren viele verschiedene Dateien und alle zusammen ergeben das Programmierprojekt. Entwicklungsumgebungen können mit solchen Projekten umgehen und alle Dateien die zum Projekt gehören, zusammen verwalten. So ist das auch mit EiffelStudio und Eiffel-Projekten. EiffelStudio verwaltet die Informationen darüber, welche Dateien alle zu einem Projekt gehören in einer speziellen Datei, deren Name mit den drei Buchstaben “ace” endet. Wir nennen diese Datei auch die *Ace-Datei*. Das Kürzel “ace” steht für *Assembly of Classes in Eiffel*, also Gruppe von Eiffel-Klassen. Im Gegensatz dazu die Dateien, die den Quelltext enthalten: die *e-Dateien* oder auch *Eiffel-Dateien*.

Wissenssicherung 1.1

Welche zwei Dateitypen gehören zu einem einfachen Eiffel-Projekt? Kannst du Dir noch andere Dateitypen vorstellen, die in einem Projekt vorkommen könnten? Überlege Dir dazu, welche anderen Dateitypen du kennst und ob diese in einem Programm, das du schreibst verwendet werden könnten.

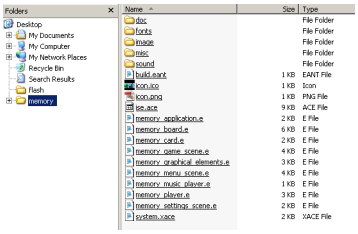


Abbildung 1.2: Projekt-Ordner

Auf dem Zusatzblatt „Aufwärmen“ siehst du wo du das Eiffel-Projekt mit dem Memory-Spiel downloaden kannst. Du kannst die Zip-Datei in deinen Bereich auf dem Computer entpacken. Es ist egal wohin du die Datei entpackst, wichtig ist nur, dass du dich später noch daran erinnerst, damit du immer am selben Projekt weiterarbeiten kannst. Wenn du das Verzeichnis öffnest, siehst du die darin enthaltenen *e-Dateien* und eine *Ace-Datei*. Du siehst aber noch andere Dateien, zum Beispiel im Unterverzeichnis *image* wo die Bilder liegen, die du dann im Memory-Spiel aufdecken wirst. Das Ganze sollte wie in der Graphik 1.2 aussehen.

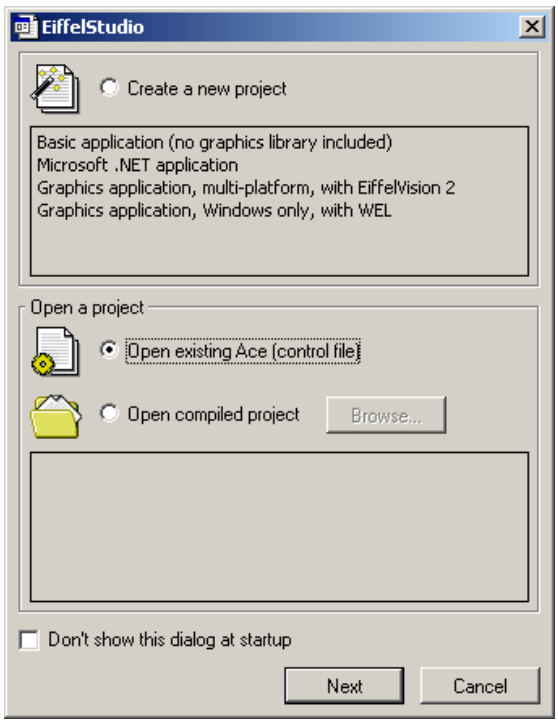


Abbildung 1.3: EiffelStudio Startdialog mit Wahl für existierende *Ace-Datei*

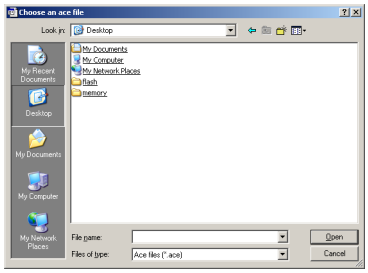


Abbildung 1.4: Auswahl der *Ace-Datei*

Du kannst nun ins EiffelStudio und mit Hilfe des Angezeigten Dialogs die *Ace-Datei* des Memory-Projekts öffnen, kompilieren und starten. Der Vorgang ist hier schrittweise mit Screenshots erläutert, damit beim ersten Mal nichts schief geht.

1. *Ace-Datei ise.ace* auswählen (siehe Abbildungen 1.3 bis 1.5)

2. In der Abbildung siehst du wie man angibt, dass das Projekt gleich nach den Öffnen kompiliert werden soll. Die Kompilierung wird eine Weile dauern, weil jetzt das ganze Projekt kompiliert wird. Der Eiffel-Quelltext wird in zwei Stufen kompiliert, dabei wird der Fortschritt zuerst mit dem Progressbar 1.7 dargestellt und danach öffnet sich ein Konsolenfenster 1.8. Das Ende der Kompilierung wird durch den Dialog 1.9 angezeigt. Keine Angst - nur die erste Kompilierung eines

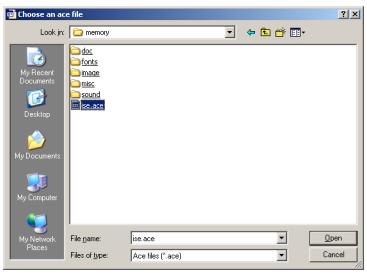


Abbildung 1.5: *Ace-Datei* ausgewählt

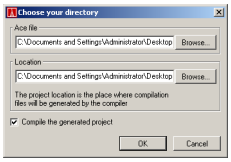


Abbildung 1.6: Projektverzeichnisse

Projekts dauert länger.

3. Nun solltest du EiffelStudio wie in Abbildung 1.10 gezeigt sehen. Mit dem markierten Button kannst du das Memory-Programm starten.

So, du hast es geschafft, du hast dein erstes Eiffel-Programm kompiliert. Nun hast du Dir eine Runde Memory gegen deinen Nachbarn verdient. Starte das Spiel wie oben beschrieben direkt aus dem EiffelStudio.

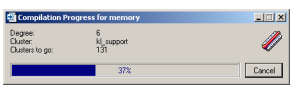


Abbildung 1.7: Kompilierungsfortschritt erste Stufe

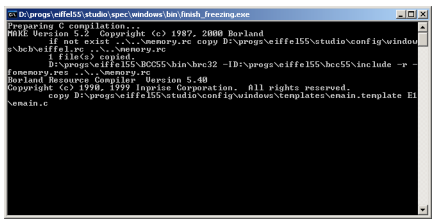


Abbildung 1.8: Kompilierungs Fortschritt zweite Stufe

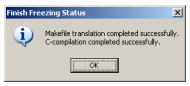


Abbildung 1.9: Kompilierung beendet

1.3 Arbeiten mit Quelltext

Nun wollen wir uns mit EiffelStudio vertraut machen. Entwicklungsumgebungen sind Programme, die von Programmierern geschrieben wurden, um anderen Programmierern das Programmieren zu erleichtern. Deshalb sind Entwicklungsumgebungen meistens sehr mächtig und enthalten viel Funktionalität, die dem Programmierer das Leben erleichtern kann. Schauen wir uns EiffelStudio erst einmal an, wie es jetzt nach der Kompilierung und nach dem Starten und wieder Beenden des Memory-Spiels aussieht. Falls du das Memory-Spiel noch nicht wieder geschlossen hast, kannst du dies auch aus EiffelStudio heraus machen - klick den Stop-Button in der Menuleiste (siehe Graphik 1.11).

Wie du in 1.12 siehst, besteht das Fenster aus verschiedenen Teilen. Beginnen wir oben links bei der Menuleiste. Da haben wir das Menu `Datei`, mit dem du ebenfalls Eiffel-Projekte öffnen kannst. Unter der Menuleiste befindet sich auf der linken Seite

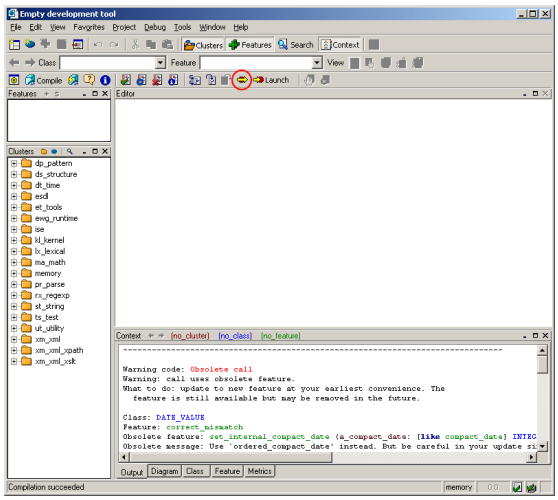


Abbildung 1.10: EiffelStudio nach beendeter Kompilierung

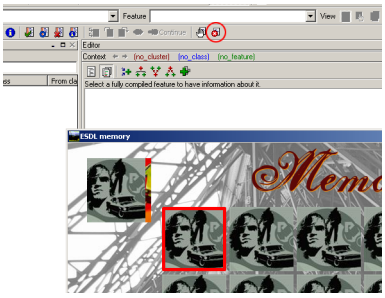


Abbildung 1.11: Stop Button

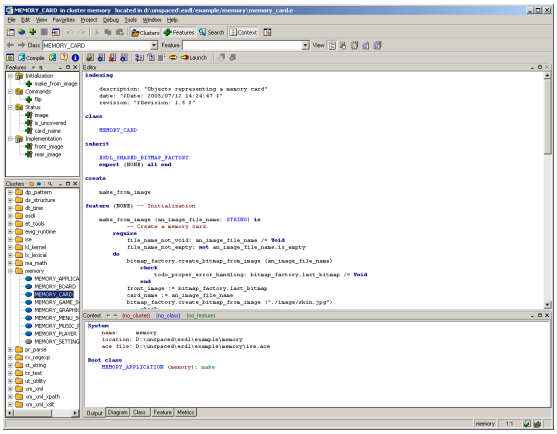


Abbildung 1.12: EiffelStudio

die Feature-Ansicht und rechts der Editor. Im Editor können wir den Quelltext aus den Eiffel-Dateien bearbeiten, in der Feature-Ansicht sehen wir die grobe Gliederung des Quelltexts, den wir im Editor geöffnet haben. Mehr zur Feature-Ansicht später.

Unter der Feature-Ansicht auf der linken Seite befindet sich die Cluster-Ansicht. Die Cluster-Ansicht zeigt alle Eiffel-Dateien, die zum Projekt gehören. Die einzelnen Dateien sind durch Ellipsen dargestellt. Die Cluster sind wie Ordner dargestellt und fassen Dateien zusammen. Du erkennst die Verzeichnisstruktur des Projekts. Meistens sieht man hier viel mehr Ordner als es in Wirklichkeit in deinem Projekt gibt. Das kommt davon, dass dein Projekt auch noch Quelltext aus Bibilotheken verwendet, die ausserhalb des Ordners deines Projektes liegen. In der Cluster-Ansicht findest du auch den Cluster, der die Eiffel-Dateien des Memory-Projekts enthält - er heisst sinngemäss “Memory”. Wie du siehst, sind alle Eiffel-Dateien, die darin vorkommen, durch eine blaue oder graue Ellipse dargestellt. Der Unterschied ist, dass die Eiffel-Dateien, die als blaue Ellipsen erscheinen, kompiliert wurden,und diejenigen, die grau sind, noch nicht. Was bedeutet das? Wie gesagt, verwenden wir grosse Bibliotheken, die von ausserhalb unserem Projekts liegen. Diese Bibliotheken enthalten zum Beispiel Funktionalität zum Darstellen von Bildern oder zum Abspielen von Musik. Wenn wir eine Applikation programmieren, die einen Teil der Funktionalität der Bibliothek gar nicht verwendet, dann muss dieser Teil der Bibliothek auch nicht mitkompiliert werden. Da-

her die unterschiedliche Darstellung in der Cluster-Ansicht. So weisst du immer ob eine der gezeigten Dateien in deinem Projekt überhaupt verwendet wird oder nicht.

Wenn du in der Cluster-Ansicht auf eine der kompilierten Eiffel-Dateien im Memory-Cluster klickst, werden sie geöffnet und im Eiffel-Editor angezeigt. Du kannst dies einmal für die Datei MEMORY_CARD tun. Wie du siehst wird sie jetzt im Editor dargestellt und ihre Struktur erscheint in der Feature-Ansicht. Mithilfe der Feature-Ansicht kannst du dich schneller zu den einzelnen Teilen der Eiffel-Datei bewegen. Klick auf *image* und beobachte wie der Editor an die Stelle springt.

Wissenssicherung 1.2

Öffne auf dieselbe Art auch noch einige andere Eiffel-Dateien. Kannst du noch andere Dinge erkennen, die beim Öffnen einer Eiffel-Datei geschehen? Findest du noch einen anderen Weg eine Eiffel-Datei zu öffnen (zum Beispiel wenn du nur ihren Namen kennst)?

Du kannst nun die Datei MEMORY_MENU_SCENE im Editor öffnen und mit Hilfe der Feature-Ansicht zu *initialize_scene* scrollen. Dein Eiffel-Editor in EiffelStudio sollte wie in Graphik 1.13 aussehen. Platziere den Cursor jetzt unter der Zeile

```
add_menu_entries_to_scene
```

und schreib:

```
start_animations
```

Damit startest du eine Animation von Memory Karten im Hauptmenu des Memory-Spiels. Damit du die Auswirkung dieser Zeile sehen kannst, musst du natürlich die Eiffel-Datei speichern (`File ▷ Save` oder `Ctrl-S`) und dann das Projekt neukompilieren und starten. Um die Neukompilierung zu starten, kannst du das Tastaturkürzel `F7` und zum starten die Kombination `Ctrl-F5` verwenden. Selbstverständlich kannst du das auch einfach mithilfe der Buttons tun, die Du vorher benutzt hast.

Wissenssicherung 1.3

Öffne die Eiffel-Datei MEMORY_GAME_SCENE und bewege dich zu *initialize_scene*. Füge nach der Zeile

```
sound_player.load_game_music
```

die Zeile

```
sound_player.play
```

ein, kompiliere das Programm neu und starte es. Wenn du alles richtig gemacht hast, sollte jetzt im Hintergrund Musik abgespielt werden, sobald du eine Memory-Partie startest.

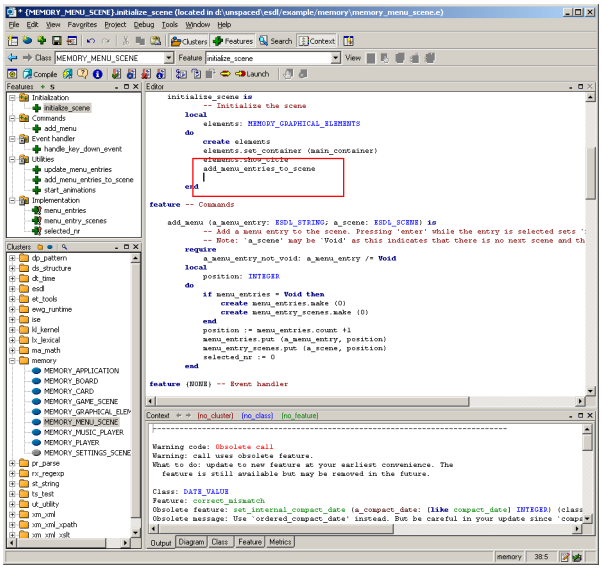


Abbildung 1.13: Editor mit geöffneter Datei

1.4 Check-Up

Im Ordner in der Mediothek findest du ein Leitprogramm das Rolf Bruderer über die Benutzung von EiffelStudio geschrieben hat. Wenn du beim durchsehen des *Kapitels 1* und *Kapitels 2* bis und mit Abschnitt *2.3 Programm zum Laufen bringen* dieses Leitprogramms das Gefühl hast, du hättest verstanden, worum es geht, dann kannst du hier noch die Antworten zu den Wissens

1.5 Antworten zu Wissenssicherungen

Lösung Wissenssicherung 1.1

- *Ace-Dateien* und *e-Dateien*
- in einem Programm, das Bilder anzeigt, könnten zum Beispiel auch noch Bild-Dateien wie *bmp-Dateien* oder auch andere Verzeichnisse verwendet werden.

Lösung Wissenssicherung 1.2

- Du öffnest andere Eiffel-Dateien, indem du sie in der Cluster-Ansicht anklickst.
- Durch Rechtsklick auf eine Eiffel-Datei in der Cluster-Ansicht kannst du diese aufheben und durch erneuten Rechtsklick über dem Editor dort fallen lassen.
- Unter der Menuleiste wird der Name, der gerade geöffneten Eiffel-Datei jeweils angezeigt, durch Eingabe des Dateinamens in diesem Feld kann man sie öffnen.

Kapitel 2

Objekte

Michela Pedroni

Übersicht

Was lernst du hier?

Wie du aus dem ersten Kapitel weisst, entsteht aus Quelltext, wenn er kompiliert wird, ein lauffähiges Programm. In der objekt-orientierten Programmierung ist der Quelltext aufgeteilt in viele Dateien. Im letzten Teil hast du den Quelltext einer ganz bestimmten Datei verändert. Wie du gesehen hast, hast du damit auch das Programm verändert. Das Ganze hat aber nur funktioniert, weil wir dir ganz genau gesagt haben, was du wo ändern musst.

Man könnte also sagen, dass du im letzten Teil noch nicht richtig programmiert hast, sondern das Tippen und das Drumherum - das Programm kompilieren und laufen lassen, EiffelStudio benützen - geübt hast. Das wollen wir aber in diesem Teil ändern! Dazu führen wir einen neuen Begriff ein: den Begriff des Objekts.

In unserem Alltag kommen wir immer wieder mit Objekten in Kontakt. Wenn du zum Beispiel zu einem Immobilienmakler gehst, dann spricht dieser von Objekten und meint damit Wohnobjekte, die zur Miete oder zum Verkauf freistehen. Auch in einer Galerie wirst du den Ausdruck Objekt (Kunstobjekt) hören. Dort meint man damit Bilder oder Skulpturen.

In der Informatik gibt es den Begriff Objekt ebenfalls. Du wirst in diesem Teil des Leitprogramms herausfinden, was Objekte sind, und wie sie das Programmieren beeinflussen. Dadurch wirst du auch besser verstehen, wieso objekt-orientierte Programmierung so heisst. Ausserdem wirst du erfahren, wie Objekte benutzt werden können um zu programmieren. Du wirst ausserdem merken, dass du bereits im ersten Teil ganz unbewusst Objekte angetroffen und benutzt hast.

Was tust du?

Zuerst liest du den Einführungstext über Objekte, die mit der Analogie der Menschen erklärt werden. Da Objekte (wie Menschen) zueinander in Beziehungen stehen, lernst du ausserdem wie solche Beziehungsnetze zeichnerisch dargestellt werden können. Ob du diese Darstellungweise anwenden kannst, wird in einer Wissenssicherung überprüft. Danach lernst du eine Methode kennen, wie Objekte einander und sich selbst Befehle geben können und welche einschränkenden Bedingungen es dabei gibt. Danach wirst du einige Worte aus dem Infortmatikfachjargon lernen. Natürlich wirst du andauernd Wissenssicherungen zu lösen haben, um das Gelernte sofort zu verinnerlichen. Das Kapitel schliesst du mit einer Check-Up-Aufgabe, die am Computer gelöst wird, ab.

Lernziele

Nachdem du dieses Kapitel durchgearbeitet hast,

- kannst du mit Hilfe einer Analogie erklären, was Objekte sind
- kannst du Beziehungsnetze von Objekten aufzeichnen
- kennst du eine Methode, um vorgegebenen Objekten Befehle zu geben

2.1 Objekte und Menschen

Du kennst Objekte aus deinem täglichen Leben. In der Informatik gibt es auch Objekte, sogenannte Softwareobjekte[1]. Ein Softwareobjekt ist ein virtuelles Ding. Dieses Ding kann etwas tun und es kann sich Sachen merken. Es kann auch mit anderen Objekten sprechen. Wenn wir eine Analogie für Objekt suchen, dann wäre wohl Mensch am passendsten[2]. Wie ein Mensch, ist ein Objekt aktiv. Man kann es etwas fragen, und es gibt Antwort. Man kann es auch dazu bringen, etwas zu tun. Dabei ist ein Objekt wirklich faul, denn es tut nur dann etwas, wenn es dazu aufgefordert wird. Ein anderes Objekt muss es darum bitten (man könnte auch sagen „ihm befehlen etwas zu tun“).

Objekte haben Namen für andere Objekte. Das ist wie bei Menschen: Sofie hat eine Mutter und wenn sie „Mutter, tu dies“ sagt, dann ist eben dieser Mensch damit gemeint. Sofie’s Vater bezeichnet denselben Menschen mit „Ehefrau“. Und wenn er sagt „Meine Ehefrau sagt ...“, dann ist es aus seiner Sicht klar, wer gemeint ist. Und Sabine nennt Sofie’s Mutter vielleicht „beste Freundin“, und wenn sie sagt „Ich gehe mit meiner besten Freundin in die Ferien“, dann weiss sie ganz genau, wer damit gemeint ist. Das ganze geht noch weiter: Sabine hat sicher auch eine Mutter und wenn sie von ihrer Mutter redet, dann meint sie einen anderen Menschen als Sofie’s Mutter. Und Sofie’s Mutter und Vater bezeichnen beide Sofie als „Tochter“. Es gibt also folgende Regeln:

- Die Bezeichnung, um einen Menschen zu identifizieren, hängt davon ab, wer darüber spricht.
- Ein Mensch kann von verschiedenen Menschen verschieden bezeichnet werden.
- Verschiedene Menschen können aber auch einen Menschen mit demselben Namen bezeichnen.

Wer nennt jetzt wen wie? So ein Beziehungsnetz ist ziemlich kompliziert! Um das Ganze besser verständlich zu machen, kannst du es aufzeichnen. Hierbei zeigt jeder Pfeil eine Beziehung zwischen zwei Menschen. Jeder Pfeil ist ausserdem mit dem Namen der Beziehung beschriftet (siehe Abbildung 2.1).

Bei Objekten ist es dasselbe. Jedes Objekt kennt eine Menge anderer Objekte und kann diese über ihren Namen ansprechen und ihnen Befehle erteilen. Es kann auch sein, dass mehrere Objekte dasselbe Objekt mit mehreren Namen bezeichnen. Und auch bei ihnen kann man Beziehungsnetze aufzeichnen.

Wissenssicherung 2.1

Gegeben ist folgendes Beziehungsnetz: Michael bezeichnet Andrea als seine „Schwester“. Michael ist der „Arzt“ von Stefan, der der „Ehemann“ von Nicole ist. Nicole wird von Andrea als ihre „beste Freundin“ bezeichnet. Und Stefan ist der „Steuerberater“ von Andrea. Zeichne das Beziehungsnetz analog zur Abbildung 2.1.

Ein objekt-orientiertes Programm besteht zumeist aus einem riesigen Beziehungsnetz mit oftmals mehreren tausend Objekten. Diese Objekte geben einander Befehle

[1] Ab jetzt meinen wir, wenn wir „Objekt“ sagen, immer „Softwareobjekt“.
[2] Die Mensch-Analogie wurde übernommen aus Bergin’s “Object-Oriented Bedtime Story” [1].

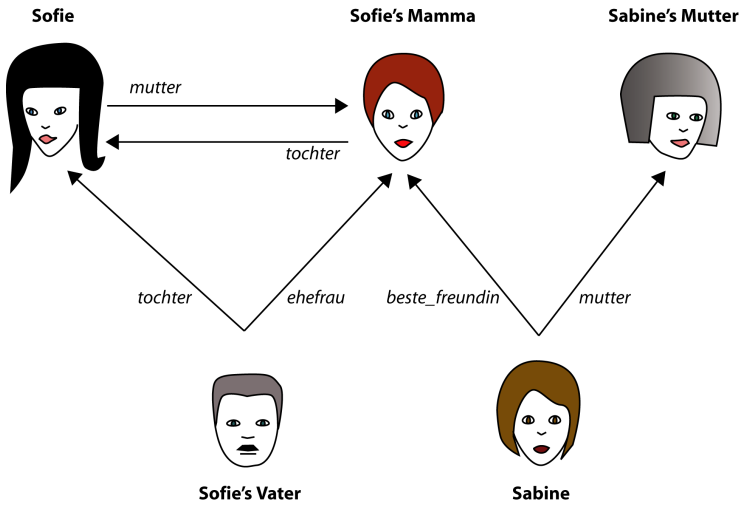


Abbildung 2.1: Sofie und ihre Beziehungen

und tauschen Informationen aus. Jedes Objekt kann natürlich nur den Objekten Befehle geben, die es kennt. Das funktioniert bei Menschen genau gleich: Sabine kann Sofie nichts befehlen, denn sie kennt sie ja nicht. Aber Sofie kann ihrer Mutter einen Befehl erteilen. Das tönt dann in etwa so: „Mutter, schau mit mir Fotos an“. Wenn Sofie's Vater es tut, tönt es so: „Ehefrau, schau doch mit mir Fotos an“ und bei Sabine „Beste Freundin, schau Dir bitte mit mir Fotos an“.

Bei Objekten ist das ganz ähnlich. Damit ein Objekt einem anderen etwas befehlen kann, muss es dieses kennen. Weil Programme immer geschrieben werden (und nie gesprochen), versucht man das ganze so kurz wie möglich zu halten. Dazu gibt es die PUNKT-NOTATION. Sofie's Befehl an ihre Mutter würde in einem Programm so geschrieben:

```
mutter . schau_fotos_an
```

Dabei steht das Zielobjekt (also das Objekt, an den der Befehl gerichtet ist) immer vor dem Punkt. Nach dem Punkt kommt der Befehl, den das Zielobjekt ausführen soll. Wie du siehst, ist der Befehl aus mehreren Worten zusammengesetzt: *schau*, *fotos* und *an*. Es gibt ganz genaue Regeln, wie diese Befehle (und auch die Zielobjektnamen) auszusehen haben, damit sie der Kompiler versteht:

Regeln für Namen
Namen, die aus mehreren Worten bestehen, werden mit ‘_‘zu einem Namen zusammengefügt. Ein Name darf nie irgendwelche Leerzeichen enthalten. Auch sonstige Sonderzeichen wie ‘-‘oder ‘\‘oder ‘&‘sind nicht erlaubt. Er darf nur aus den Buchstaben a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z und A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z und eben ‘_‘bestehen. Wie du siehst sind auch Umlaute (ä, ö, ü) nicht erlaubt. Diese Regeln gelten für alle Namen, die in einem Programm vorkommen.

Die allgemeine Form eines Befehls an ein Zielobjekt sieht so aus:

```
ziel_objekt_name . tu_etwas
```

Wissenssicherung 2.2

Schreibe die folgenden Befehle mit der Punkt-Notation auf. Dazu musst du zuerst herausfinden, wer den Befehl erteilt und welchen Namen er/sie für das Zielobjekt verwendet. Benutze dazu die Beziehungsnamen aus der Abbildung 2.1. Überlege Dir auch, ob der verlangte Befehl überhaupt möglich ist.

1. *Sofie's Vater befiehlt Sofie's Mamma, sie solle Fotos anschauen.*
2. *Sabine befiehlt Sofie's Mamma, sie solle Fotos anschauen.*
3. *Sabine befiehlt ihrer Mutter, sie solle einen Brief schreiben.*
4. *Sabine's Mutter befiehlt Sofie, sie solle einen Kuchen backen.*

Vielleicht ist dir schon aufgefallen, dass eine der Zeilen, die du in Kapitel 1 eingetippt hast, auch der Punkt-Notation folgt. Und auch dort hast du einem Zielobjekt einen Befehl gegeben. Schauen wir uns den Code nochmals an:

```
music_player . play
```

Hier haben wir also ein Zielobjekt: *music_player*. Diese Zeile befiehlt dem *music_player*, er solle Musik abspielen. Wie du gesehen hast, hat er das auch getan.

Die Frage ist nun: Welches Objekt gibt dem *music_player* in den obigen Zeilen den Befehl Musik abzuspielen? In unserem Programm gibt es Szenen. Die erste Szene ist die, in der du zwischen Start und Quit wählen kannst. Diese Szenen nennen wir fortan Menuszene. Du siehst sie, wenn du das Programm frisch aufstartest (siehe Abbildung 2.2). Die zweite Szene unseres Spiels nennen wir jetzt mal Gameszene und du siehst sie, wenn du auf Start in der Menuszene geklickt hast (siehe Abbildung 2.3). In dieser Szene wird die Musik abgespielt.

Die Menuszene und die Gameszene sind Objekte. Wie die Gameszene aussieht, wird in der Datei memory_game_scene.e beschrieben. Und eben diese Datei haben wir verändert. Dieses Objekt befiehlt anderen Objekten (so wie dem *music_player*) Sachen zu tun (zum Beispiel Musik abzuspielen).

Wir können auch hier ein Beziehungsnetz aufzeichnen. Dabei zeichnen wir die Objekte als Vierecke:

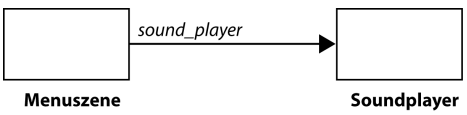


Abbildung 2.2: Menuszene des Memory-Spiels

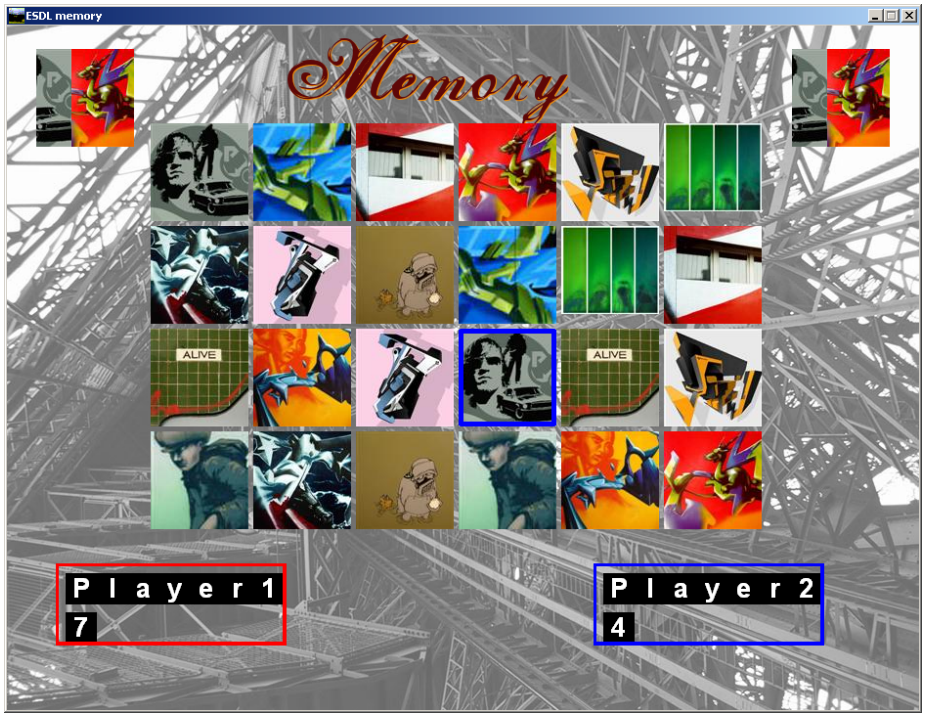


Abbildung 2.3: Gameszene des Memory-Spiels

2.2 Einschränkungen beim Befehle erteilen

Du hast also jetzt erfahren, dass Objekte einander Befehle geben können. Die erste Einschränkung hast du schon mitbekommen, nämlich dass ein Objekt das Zielobjekt kennen muss. Es gibt eine zweite Einschränkung: Nicht jedes Objekt, kann jeden beliebigen Befehl ausführen. Pro Objekt ist ganz genau festgelegt, welche Anfragen man stellen kann.

Geh zu einem der Computer und starte EiffelStudio. Öffne das Memory-Projekt, das du im letzten Kapitel erstellt hast. Öffne die Eiffel-Datei MEMORY_GAME_SCENE und gehe nochmals an den Ort, wo du im letzten Kapitel deine Änderungen gemacht hast (*initialize_scene*. Was du dort also getan hast, war einem anderen Objekt einen Befehl zu geben. Tippe nun *music_player.iss_ein_gipfeli* ein. *iss_ein_gipfeli* ist ein Befehl, den das Objekt mit dem Namen *music_player* nicht bearbeiten kann. Kompiliere nun erneut. Was steht in der Fehlermeldung? Was passiert, wenn du statt *music_player* *sound_player* eingibst (das ist ein Name für den kein Zielobjekt bekannt ist)?

Abbildung 2.4 zeigt die Fehlermeldung im Fenster, wenn du den unbekannten Befehl *iss_ein_gipfeli* eingibst und Abbildung 2.5 zeigt die Fehlermeldung für das unbekannte Zielobjekt *music_player*. Bei beiden Fehlern sieht die Fehlermeldung fast gleich aus, weil in beiden Fällen einfach etwas Unbekanntes eingegeben wurde.

```
Error code: VEEN
Error: unknown identifier.
What to do: make sure that identifier, if needed, is final name of
feature of class, or local entity or formal argument of routine.
```

Diese Fehlermeldung sagt aus, dass der IDENTIFIER (der Fachbegriff für Namen von Befehlen oder bekannten Objekten) nicht bekannt ist. Was man also tun muss, ist sich vergewissern, ob ein Befehl mit diesem Namen wirklich auf das Zielobjekt anwendbar ist oder ob ein Zielobjekt mit diesem Namen wirklich existiert. Die nächsten Zeilen geben mehr Informationen darüber, wo der Fehler aufgetreten ist:

```
Class: MEMORY_GAME_SCENE
Feature: initialize_scene
Identifier: iss_ein_gipfeli
Taking no argument
Line: 55
      music_player.play
->    music_player.iss_ein_gipfeli
      player1.show
```

Die erste Zeile zeigt uns, dass der Fehler in der Datei MEMORY_GAME_SCENE passiert ist und zwar bei *initialize_scene*. Die dritte Zeile gibt uns den fehlerhaften Namen (Identifier) an.

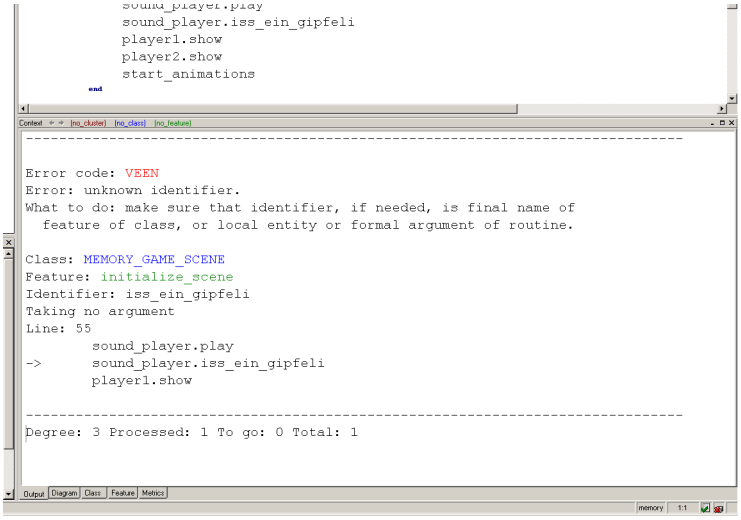


Abbildung 2.4: Fehlermeldung bei unbekanntem Befehl

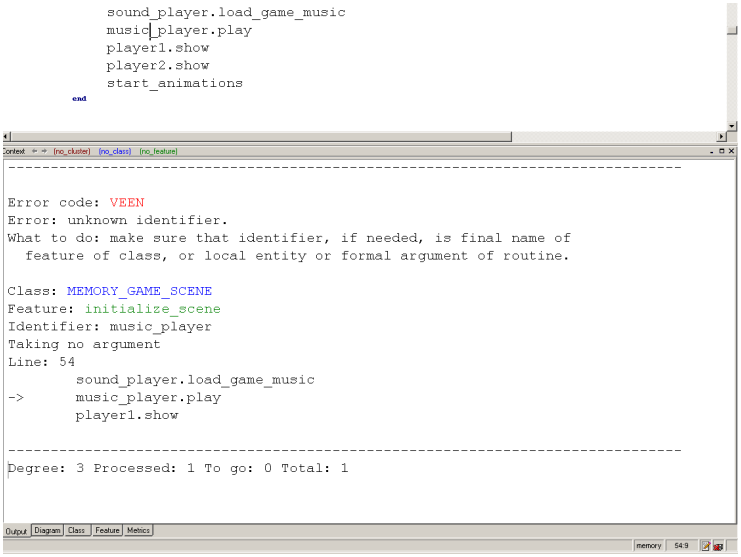


Abbildung 2.5: Fehlermeldung bei unbekanntem Zielobjekt

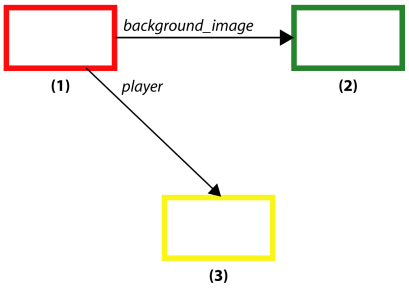


Abbildung 2.6: Beziehungsnetz

Wissenssicherung 2.3

In Abbildung 2.6 siehst du ein weiteres Beziehungsnetz. Dabei geht vom rot umrahmten Objekt (1) ein Beziehungspfeil mit Namen background_image (übersetzt: Hintergrundbild) zu einem weiteren Objekt (2) mit grüner Umrahmung und ein zweiter Pfeil *player* (übersetzt: Spieler) zu einem dritten Objekt (3) mit gelbem Rahmen. Nimm nun an, dass das Objekt (2) folgende Befehle ausführen kann: *show* (übersetzt: sich zeigen), *zoom_in* (übersetzt: vergrössern) und *hide* (übersetzt: sich verstecken). Objekt (3) kann folgenden Befehl ausführen: *set_color_black* (übersetzt: die Farbe auf schwarz setzen). Beantworte folgende Fragen aus Sicht des roten Objekts (1).

1. Welchen Zielobjekten kann das rot umrandete Objekt (1) Befehle geben?
2. Wie kann das Objekt (1) das Hintergrundbild ausblenden (verstecken)? Schreibe den Code auf.
3. Kann das rote Objekt (1), dem Spieler befehlen sich auszublenden?
4. Wie kann das rot umrandete Objekt (1) beim Spieler die Farbe schwarz setzen?
5. Welche der folgenden Befehle sind korrekt?
   (a) *player.show*
   (b) *player.hide*
   (c) *background_image.hide*
   (d) *background_image.set_color_black*
   (e) *player.zoom_in*
   (f) *background_image.zoom_in*

2.3 Ein Objekt befiehlt sich selbst

Nun hast du gesehen, dass ein Objekt anderen Objekten mit Hilfe der Punktnotation Befehle geben kann. Dasselbe gilt auch für sich selbst. Ein Objekt kann sich selbst Befehle geben. (Auch Menschen können das. Du kannst zum Beispiel dir selbst befehlen, aufzuhören auf dem Stift zu kauen.)

Die grosse Frage ist nun, wie sie das tun. Dazu müsste ein Objekt ja einen Namen für sich selbst haben. Und, wirklich, das gibt es. Wann immer ein Objekt von sich selbst spricht, dann sagt es **Current**. Es kann also mit Hilfe der Punktnotation sich selbst Befehle geben.

Nehmen wir an, dass das Objekt Gameszene (aus dem vorletzten Abschnitt) den Befehl *show_title* kennt. Dann kann es sich selbst befehlen, den Titel anzuzeigen, indem folgender Befehl eingegeben wird:

Current. *show_title*

Im Allgemeinen kann sich jedes Objekt selbst etwas befehlen, vorausgesetzt es kann den Befehl ausführen. Da sich jedes Objekt immer selbst kennt, musst du dir darüber keine Gedanken machen. Die allgemeine Form eines Befehls, dass ein Objekt sich selbst gibt, sieht also so aus:

Current.*tu_etwas*

Es gibt eine Abkürzung, um sich selbst Befehle zu geben. Die Programmiersprache Eiffel, nimmt einfach an, wenn kein Zielobjekt angegeben ist, dass dann ein Selbstbefehl gegeben wird. Wir können also **Current**.*show_title* mit *show_title* abkürzen und unser Programm tut genau dasselbe. Im allgemeinen ist also

tu_etwas

ein Selbstbefehl.

Wissenssicherung 2.4

Nimm an ein Objekt kennt die folgenden Befehle: *set_black_color*, *show* und *show_background_image*. Gib jeweils beide Formen an, die das Objekt benutzen kann, um die Befehle sich selbst zu geben.

2.4 Softwareobjekte und Fachjargon

Bis jetzt hast du zwei Sachen kennengelernt, die ein Softwareobjekt ausmachen: Die Befehle, die es ausführen kann und die Objekte, die es kennt.

Die Objekte, die ein Softwareobjekt kennt, definieren seinen Zustand; es sind also die Eigenschaften eines Objekts. Ein Spieler in unserem Memory-Spiel kennt zum Beispiel ein Objekt, das seinen Namen beinhaltet, ein weiteres, das den Punktestand repräsentiert und ein Farbobjekt, das dem Spieler eine Farbe zuordnet. Diese Objekte werden mittels den Bezeichnungen *name*, *points* und *color* angesprochen. Und jedes

dieser Objekte kann weitere Objekte ansprechen, so zum Beispiel das Namenobjekt die Objekte, die die einzelnen Buchstaben beinhalten. Nun wird auch klar, weshalb diese Objekte den Zustand des Spieler-Objekts verändern: Wenn sich eines der ansprechbaren Objekte ändert, dann ändert sich das Spieler-Objekt indirekt.

Du hast bereits erfahren, dass ein Objekt, um ein anderes Objekt anzusprechen, eine Beziehung zu diesem Objekt haben muss. Diese Beziehungen haben Namen; wie zum Beispiel die drei Beziehungen des Memoryspieler-Objekts *name*, *points* und *color*. Diese Beziehungen nennt man im Informatikfachjargon QUERIES[3]..

Definition Query
Eine Query bietet Zugriff auf bekannte Objekte und ermöglicht einen Aspekt des Zustandes eines Objekts abzufragen.

Die zweite Sache, die du kennengelernt hast ist, wie man einem Objekt Befehle erteilt. Für jedes Objekt ist genau definiert, welche Befehle es ausführen kann. Für das Memoryspieler-Objekt ist das auch so: dieses Objekt kann (unter anderem) die folgenden Befehle ausführen: *show* zeichnet die Spielerinformationen wie Name und Punktestand auf den Bildschirm und *increase_points* erhöht den Punktestand um 1. Diese Befehle definieren, wie der Zustand eines Objekts verändert werden kann. Der Fachbegriff für Befehl ist COMMAND[4]..

Definition Command
Die Menge der Commands eines Objekts bestimmt, wie sein Zustand verändert werden kann.

Queries und Commands bestimmen das Verhalten und das Wesen eines Objekts. Als Sammelbegriff für Queries und Commands benutzen wir den Begriff FEATURES[5].

Definition Feature
Die Features eines Objekts sind alle seine bekannten Queries und die anwendbaren Commands. Die Features eines Objekts bestimmen sein Verhalten und seine Merkmale.

Wenn du im Quelltext ein Objekt über einen Query ansprichst, oder einen Befehl erteilst, dann nennt man das einen Aufruf. Auf Englisch heisst das ein FEATURE CALL. Es ist also so, dass die bei einem Objekt vorhandenen Queries und Commands Features heissen, und die Benutzung der Features nennt man Feature calls. *player*.*increase_points* ist zum Beispiel ein Feature call.

Definition Feature call
Wenn ein Feature auf ein Objekt angewandt wird, dann nennt man das einen Feature call.

Alle Features haben einen Namen über den sie angesprochen werden können. Beim Memoryspieler sind das *name*, *points* und *color* für seine Queries. *show* und *increase_points* sind die Namen für seine Commands. Beachte das diese Bezeichnungen

[3]Query: Englisch für Abfrage
[4]Command: Englisch für Befehl
[5]Feature: englisch für Eigenschaft/Merkmal/Fähigkeit.

immer den Regeln auf Seite 24 folgen müssen. Diese Regeln sind zwingend - wenn du sie nicht einhälst, dann gibt es einen Compilerfehler.

Es gibt zusätzliche Regeln, die aber keinen Compilerfehler zur Folge haben, wenn du sie nicht einhälst. So werden im allgemeinen für Commands Namen benutzt, die mit einem englischen Verb beginnen (wie *show*, *increase_points*. Die Namen der Queries sind normalerweise englische Nomen (wie *points*, *name* oder *color*. Auch sie können aus mehreren Teilen bestehen (zum Beispiel *first_name* und *last_name*).

Wissenssicherung 2.5

Versuche anhand der obigen Regeln herauszufinden, welche der folgenden Features Queries und welche Commands sind[6].

1. *make_with_size*
2. *draw*
3. *width*
4. *height*
5. *match*
6. *deal*
7. *move_cursor*
8. *cursor_color*
9. *uncover*
10. *cards*

2.5 Check-Up

Nun hast du alles erfahren, um einer ersten Herausforderung stand zuhalten. Dieser Abschnitt besteht nur aus einer Aufgabe, die du nun mit dem Wissen, dass du bis jetzt erlangt hast, lösen können solltest. . .

Öffne die Datei MEMORY_GAME_SCENE des Memory-Projekts und bewege dich zum Feature *show_game_over*. Dieses Feature wirst du nun füllen. Dir stehen dabei folgende Objekte mit den auf sie anwendbaren Features zur Verfügung:

- *board* mit Features *deal*, *match* und *uncover*
- *player1* mit Features *increase_points* und *show_player_image*
- *player2* mit Features *increase_points* und *show_player_image*
- *music_player* mit Features *load_game_music*, *load_game_over_music*, *play* und *stop*

[6] Diese Features sind für das Memorybrettobjekt vorhanden.

- *game_over_screen* mit Features *show* und *hide*

Füge nach der Zeile

```
−− FILLED IN BY STUDENTS −−
```

eine neue Zeile ein (durch Drücken der Tastaturtaste Enter). Danach gibst du vier Befehle ein. Jeder dieser Befehle muss auf einer neuen Zeile stehen.

1. Der erste Befehl zeigt den *game_over_screen*. Schaue in der obigen Liste nach, welches Objekt dein Zielobjekt sein könnte und überlege dir danach, welches der richtige Befehl ist. Tippe die Anweisung mit der Punktnotation ein. Kompiliere dein Programm. Falls eine Fehlermeldung erscheint, dann überprüfe, ob du am richtigen Ort bist, und überprüfe, ob du alles richtig geschrieben hast. Versuche dann erneut zu kompilieren. Falls das immer noch nicht klappt, dann frage einen Mitschüler oder deinen Lehrer.

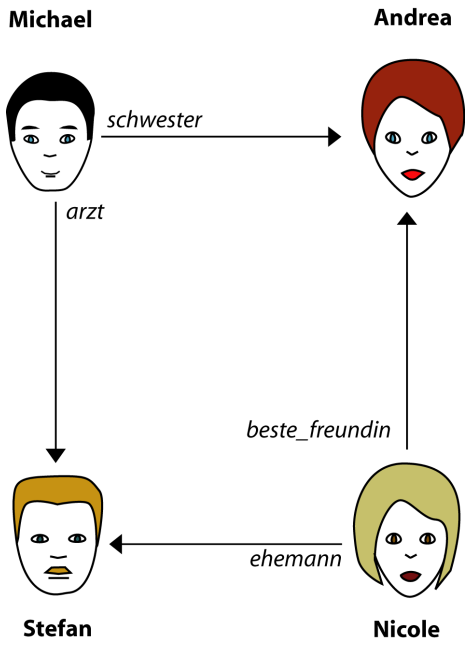
2. Der zweite Befehl soll die Musik anhalten. Mache dasselbe wie beim ersten Befehl. Du solltest erneut kompilieren und bei Fehlermeldungen korrigieren, bis es läuft.

3. Der dritte Befehl soll die Musik laden, die bei “Game over” abgespielt werden soll. Schaue wiederum oben nach, welches dein Zielobjekt sein könnte und wie der richtige Befehl heisst.

4. Der letzte Befehl spielt die geladenen Musik nun ab. Vergiss nicht nochmals zu kompilieren. Teste nun dein Programm, indem du es startest und einmal zu Ende spielst.

2.6 Antworten zu Wissenssicherungen

Lösung Wissenssicherung 2.1



Lösung Wissenssicherung 2.2

Anmerkung: Wenn der genaue Name des Befehls von dem der Lösung abweicht, dann macht das nichts (zum Beispiel statt *schreib_einen_brief* *schreibe_brief*). Überprüfe jedoch, dass du die Regeln für Namen auf Seite 24 befolgst!

1. *ehefrau*.*schau_fotos_an*
2. *beste_freundin*.*schau_fotos_an*
3. *mutter*.*schreib_einen_brief*
4. Funktioniert so nicht, weil Sabine's Mutter Sofie nicht kennt!

Lösung Wissenssicherung 2.3

1. Objekten (2) und (3) (gelb und grün umrandet).
2. *background_image*.*hide*
3. Nein, denn Objekt (3) kennt den Befehl zum ausblenden (oder verstecken) nicht.
4. *player*.*set_color_black*

5. Korrekt sind (c) und (f). Alle anderen würden eine Kompilerfehler erzeugen.

Lösung Wissenssicherung 2.4

Current.*show_background_image* und *show_background_image*
Current.*set_black_color* und *set_black_color*
Current.*show* und *show*

Lösung Wissenssicherung 2.5

1. Command
2. Command
3. Query
4. Query
5. Unklar, weil *match* sowohl Verb als wie auch Nomen ist (hier ist es aber ein Command).
6. Unklar, weil *deal* sowohl Verb als wie auch Nomen ist (hier ist es aber ein Command).
7. Unklar, weil *move* sowohl Verb als wie auch Nomen ist (hier ist es aber ein Command).
8. Query
9. Command
10. Query

Du siehst also, dass diese Regeln helfen können, um herauszufinden, ob etwas ein Zielobjekt bezeichnet oder einen Befehl. Wenn du nun aber denkst, dass dies nicht gut genug ist, dann hast du recht. Im nächsten Kapitel wirst du eine bessere Methode lernen, um herauszufinden, welches Feature eine Query und welches ein Command ist.

Kapitel 3

Klassen

Michela Pedroni

Übersicht

Was lernst du hier?

Objekt-orientierte Computerprogramme bestehen aus vielen (oftmals mehreren tausend) Objekten, die einander Befehle erteilen. Bis jetzt hast du gelernt, Objekte mit Hilfe der Analogie der Menschen zu verstehen. Du hast auch schon einige Objekte im Memory-Programm gesehen, so zum Beispiel den *player*, das *background_image* oder die Gameszene. Natürlich gibt es im Memory-Spiel noch viel mehr Objekte, so zum Beispiel das Memoryboard, jede der Memorykarten, die Spieler, die Szenen, den Titel einer Szene, und so weiter... Alle diese Objekte haben Features: Commands, die aufgerufen werden können um das Zielobjekt zu verändern, und Queries, um auf weitere Objekte zuzugreifen.

Wenn wir uns nun diese Objekte genauer anschauen, dann sticht etwas sofort ins Auge: Das Memoryboard ist grundverschieden von den Memorykarten, aber die einzelnen Memorykarten ähneln sich. So liegt die Annahme nahe, dass die Memorykarten die gleichen Befehle ausführen können und auch dieselben Arten von Beziehungen haben. Und tatsächlich, so ist es. Das liegt daran, dass die Memorykarten alle zur selben Klasse von Objekten gehören.

Klassen sind die zentralen Bausteine der objekt-orientierten Programmierung. Im letzten Kapitel hast du gelernt, wie ein Objekt mit Hilfe der Punktnotation andere Objekte dazu bringen kann, Befehle auszuführen. Ausserdem hast du gelernt, dass ein Objekt auch sich selbst Befehle erteilen kann. Dabei gab es aber zwei Einschränkungen:

1. Das befehlende Objekt, muss das Zielobjekt kennen.
2. Das Zielobjekt muss den Befehl ausführen können.

Aus diesen Bedingungen stellen sich zwei Fragen:

1. Wie kannst du herausfinden, ob das befehlende Objekt das Zielobjekt kennt?

2. Wie kannst du herausfinden, welche Befehle das Zielobjekt ausführen kann?

Mit Hilfe von Klassen lassen sich diese zwei Fragen beantworten.

Was tust du?

Lies zuerst die Lernziele durch. Nimm dir dabei Zeit, es ist wichtig dass du verstehst, um was es geht. Danach liest du den ersten Abschnitt dieses Kapitels durch und prüfst mit der Wissenssicherung, ob du den Unterschied zwischen Klasse und Objekt verstanden hast. In den darauf folgenden Abschnitten geht es darum, die Beschreibung einer Klasse lesen und verstehen zu lernen. In diesem Teil wurden meist etwas vereinfachte Klassenbeschreibungen benutzt, da die echten (die im Programm benutzt werden) noch zu kompliziert sind. Auch hier helfen dir die Wissenssicherungen herauszufinden, ob du den Abschnitt begriffen hast. Anschliessend nimmst du dir eine unveränderte Klassenbeschreibung vor und identifizierst die Teile, die du nun schon verstehst. Zu guter letzt bei der Check-Up-Aufgabe wendest du das Gelernte natürlich im Memory-Projekt direkt an.

Lernziele

Nachdem du dieses Kapitel durchgearbeitet hast,

- kannst du erklären, was eine Klasse ist und kennst den Unterschied zwischen Klasse und Objekt
- kannst du für ein beliebiges Zielobjekt herausfinden, welche Befehle aufgerufen werden können
- weisst du, wie neue Objekte erzeugt werden, und kannst dies selbstständig tun

3.1 Von Objekten zur Klasse

Eine Klasse beschreibt die Commands und Queries, die die Objekte dieser Klasse haben. Ein Objekt ist also durch die Klasse, zu der es gehört, charakterisiert. Jedes Objekt wird durch eine Klasse beschrieben, ganz egal ob es nun zehn, zweihundert oder nur genau ein Objekt davon gibt.

Definition Klasse
Eine Klasse ist die Beschreibung einer Menge von Objekten, die dieselben Features anbieten.

Eine Klasse repräsentiert also eine Kategorie von Dingen; und ein Objekt repräsentiert ein bestimmtes Ding. Der Zusammenhang zwischen Klasse und Objekt ist der übliche, wie du ihn auch im Alltag kennst: „STUHL“ ist eine Kategorie und der Stuhl, auf dem du im Moment sitzt, ist ein Element dieser Kategorie.

Definition Instanz, erzeugende Klasse
Wenn ein Objekt O eines der Objekte ist, die durch die Klasse K beschrieben werden, dann nennt man O eine INSTANZ der Klasse K. Die Klasse K heisst dann die ERZEUGENDE KLASSE von Objekt O.

Definition Typ
Wenn die Klasse K die erzeugende Klasse eines Objekts O ist, dann sagt man, dass O den TYP K hat.

Klassen beschreiben also eine Menge von Objekten. Sie sind die Baupläne, nach denen Objekte gebaut werden und funktionieren. Die Klassen sind auch das, was eigentlich programmiert wird. Die Änderungen, die du bis jetzt am Programm erzeugt hast, wurden gemacht, indem du bestehende Klassen verändert hast. Du hast also nicht direkt einzelne Objekte verändert, sondern den Bauplan und damit das Verhalten dieser Objekte, so wie es in der Klasse definiert war.

Wo sind denn nun die Objekte? Die Antwort ist einfach: Meistens nicht da. Die Objekte treten erst auf, wenn du das Programm startest. Nach Beendigung des Programms verschwinden sie wieder. Objekte sind also nur zur LAUFZEIT vorhanden.

Definition Laufzeit
Unter der LAUFZEIT versteht man in der Informatik die Zeitspanne, innerhalb der ein Programm ausgeführt wird.

Die Arbeit eines Programmierers ist ähnlich wie die eines Architekt’s, der Häuser entwirft und baut: Zuerst wird der Bauplan entworfen, danach wird das Haus gebaut. Im Falle des Programmierers werden zuerst die Klassen geschrieben. Dabei stellt sich der Programmierer immer die Objekte vor, die sich beim Ablauf des Programms gegenseitig Befehle geben. Danach erst kann das Programm ausgeführt werden. Einen Vorteil hat der Programmierer im Gegensatz zum Architekten: der Programmierer kann sein Programms so oft laufen lassen, wie er will, und auch nachträglich noch Änderun-

gen einbauen[1]. Das ist im allgemeinen auch die Vorgehensweise, die sich am meisten bewährt: Immer kleine Teile des Programms erstellen oder ändern und danach sofort ausprobieren, ob es so funktioniert wie geplant.

Wissenssicherung 3.1

Überlege dir, wie die Beziehung zwischen Objekt und Klasse gestaltet ist. Beantworte dazu die folgenden Fragen:

1. *Welche Anzahl von Klassen wird benötigt, um ein Objekt zu beschreiben?*
2. *Wie viele Objekte können von einer Klasse beschrieben werden?*

Wissenssicherung 3.2

Versuche eine eigene Analogie zu finden, die den Unterschied zwischen Klasse und Objekt gut erklärt.

3.2 Leseübung

Nun aber zurück zur Frage, wie du herausfinden kannst, welche Queries und Commands für ein Objekt vorhanden sind. Dazu musst du Klassentexte lesen lernen.

Sehen wir uns die Klasse MEMORY_CARD (Listing 3.1) genauer an[2]. Diese Klasse beschreibt wie die Memory-Spieler-Objekte in unserem Programm auszusehen haben.

1,2: Die ersten zwei Zeilen beschreiben auf Englisch, welche Art von Objekten diese Klasse beschreibt. In unserem Fall sind das Objekte, die eine Memorykarte repräsentieren. Merke dir, dass wir, wenn wir über Objekte sprechen, immer Softwareobjekte meinen und nicht die physikalischen Objekte (also nicht die wirklichen Memorykarten, mit denen du zu Hause spielst, sondern nur abstrakte Repräsentationen, mit denen wir programmieren können).

4, 5, 26: Das Schlüsselwort **class** zeigt an, dass nun die Klassenbeschreibung folgt. Der darauf folgende Name sagt uns, um die Beschreibung wessen Klasse es sich handelt. Hier ist das die Klasse MEMORY_CARD. Es spielt keine Rolle, ob der Klassenname auf einer neuen Zeile ist. Er muss jedoch durch ein oder mehrere Leerzeichen und/oder einer oder mehrerer neuen Zeilen und/oder einem oder mehrerer Tabulatoren vom Schlüsselwort **class** getrennt sein. Klassennamen werden immer gross geschrieben. Auf Zeile 26 findest du das Schlüsselwort **end**, dass die Klassendefinition beendet. **class** muss immer ein **end** als Gegenstück am Ende der Klasse haben.

[1] Für einen Architekten wäre diese Vorgehensweise zu kostspielig.

[2] Wenn du die Datei MEMORY_CARD im EiffelStudio öffnest, dann wirst du eine etwas andere Klassenbeschreibung sehen. Wir haben hier Zeilen, die im Moment noch unwichtig sind, weggelöscht.

Listing 3.1: Klasse MEMORY_CARD

```
1  indexing
2      description : Objects representing a memory card
3
4  class
5      MEMORY_CARD
6
7  feature -- Commands
8
9      flip is
10             -- Flip the card.
11         do
12             -- Something here
13         end
14
15 feature -- Queries
16
17     card_name: STRING
18             -- Name of card
19
20     image: EM_BITMAP
21             -- The displayed image of the memory card
22
23     is_uncovered : BOOLEAN
24             -- Was this memory card uncovered in the current turn?
25
26 end
```

7, 15: Hier sehen wir ein weiteres Schlüsselwort: **feature**. Dieses Wort zeigt an, dass nun Features beschrieben werden. Das was dahinter steht, ist ein Kommentar und wird vom Compiler nicht gebraucht, um das Programm zu erzeugen. Es gibt uns aber die Möglichkeit, die Features zu gruppieren. So werden hier in diesem ersten Abschnitt Commands beschrieben und ab Zeile 15 alle Queries. Du könntest aber auch einfach alle Features unter das Schlüsselwort **feature** schreiben, das -- Commands weglöschen und die Zeile 15 auch, und dein Programm würde noch gleich funktionieren. Nur wäre es dann natürlich weniger übersichtlich.

9: *flip* ist der Name des ersten Befehls, den Objekte der Klasse der MEMORY_CARDs uns zur Verfügung stellen. Danach folgt das Schlüsselwort **is**. Dieses gibt uns an, dass nun programmiert wird, was der Command *flip* tut.

10: Hier steht wiederum ein Kommentar. Das ist wieder Text, der nicht nötig ist um das Programm auszuführen, sondern dazu da ist, einem Menschen (dem Programmierer) zu helfen zu verstehen, was das Feature tut. Du kannst an jeder beliebigen Stelle im Klassentext Kommentare einfügen. Schreibe einfach „--“ und deine Anmerkung dahinter. Merke dir aber, dass du die „--“ auf jeder neuen Zeile wieder schreiben musst, wenn du mehrzeilige Kommentare machen möchtest.

11 - 13: Das Schlüsselwort **do** deutet an, dass nun durch Aufrufen von anderen Befehlen bei möglicherweise anderen Zielobjekten das Feature *flip* definiert wird. Das Schlüsselwort **end** auf Zeile 13 zeigt an, dass diese Definition hier endet. Was du bis jetzt programmiert hast, waren immer Zeilen, die du zwischen einem **do** und einem **end** in einem Feature geschrieben hast.

17, 20, 23: Hier kommen unsere Queries! Wie beim Command wird zuerst der Name der Query angegeben (so zum Beispiel *card_name* auf Zeile 17). Danach folgt ein Doppelpunkt und ein weiterer Klassenname. Dieser Klassenname (im Beispiel für *card_name* ist das STRING) gibt uns die Information, welchen Typ unser Zielobjekt hat. Wir können also dem Objekt, das durch die Query *card_name* angesprochen werden kann, die Befehle geben, die im Klassentext von STRING angegeben werden.

18, 21, 24: Das sind wiederum Kommentare und beschreiben, welche Beziehung die Query repräsentiert.

Wie erkennst du, dass ein Feature eine Query und kein Command ist? Das tust du am besten, indem du die erste Zeile der Featuredeklaration anschaust. Bei Queries hast du am Ende der Zeile immer einen Doppelpunkt und einen Klassenamen. Der Klassenname zeigt dir, welche Art von Objekt du über die Query ansprechen kannst. Das brauchst du bei Commands nicht und wirst es auch nicht finden.

Nun hast du dir bereits einen grossen Teil des Wissen, den du brauchst, um einen Klassentext zu verstehen, angeeignet. Nun wollen wir unser Vokabular aber noch etwas erweitern. Gewisse dieser Begriffe hast du bereits in den obenstehenden Erklärungen gesehen. Auf der Abbildung **??** siehst du am Beispiel unserer MEMORY_CARD-Klasse, wie die einzelnen Teile des Klassentext's genannt werden.

① Klassendeklaration
② Klassenname
③ Featuredeklaration
④ Featurename
⑤ Kommentar
⑥ Featurebody

Wissenssicherung 3.3

Schau dir den Klassentext in Listing 3.2 an und beantworte die Fragen.

1. *Welche Klasse beschreibt dieser Klassentext?*
2. *Welche Features werden deklariert?*
3. *Welche Klassennamen kommen darin vor?*
4. *Welche der Features sind Queries?*
5. *Welche der Features sind Commands?*
6. *Welche Zeilen gehören zum Featurebody von Feature increase_points?*
7. Welche Zeilen gehören zum Featurebody von Feature *points*?

Listing 3.2: Klasse MEMORY_PLAYER

```
1  class
2      MEMORY_PLAYER
3
4  feature
5
6      show is
7              -- Show the player.
8          do
9              -- Do something.
10         end
11
12     name: STRING
13             -- Name of the player
14
15     points : INTEGER
16             -- Current number of points of the player
17
18     increase_points is
19             -- Increase ' points ' by one.
20         do
21             -- Do something.
22         end
23
24     reset is
25             -- Reset score for new game.
26         do
27             -- Do something.
28         end
29
30     color : EM_COLOR
31             -- Color of player
32
33 end
```

Abbildung 3.1: Teile eines Klassentexts

```
class                                                                    ①
    MEMORY_CARD ②

feature -- Commands

    flip ④ is                                                            ③
            -- Flip the card.⑤
        do                                                               ⑥
            -- Do something
        end

feature -- Queries

    card_name ④ : STRING ②                                               ③
            -- Name of card ⑤

    image ④ : EM_BITMAP ②                                                ③
            -- The displayed image of the memory card ⑤

    is_uncovered ④ : BOOLEAN ②                                           ③
            -- Was this memory card uncovered in the current turn? ⑤

end
```

8. Auf welchen Zeilen findest du einen Kommentar?

3.3 Export status

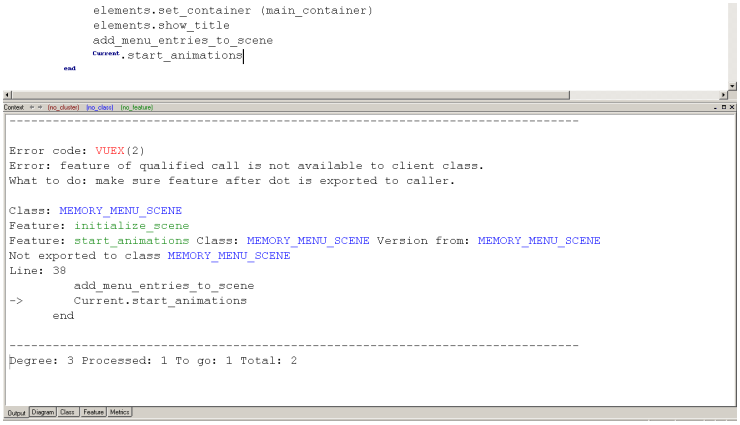
In Kapitel 1.3 auf Seite 18 hast du den Selbstbefehl

> *start_animations*

in der Klasse MEMORY_MENU_SCENE in das Feature *initialize_scene* eingefügt. Im zweiten Kapitel hast du gelernt, dass dies eine Abkürzung für den folgenden Befehl ist:

> **Current**. *start_animations*

Starte EiffelStudio und ändere *start_animations* in **Current**.*start_animations* um. Kompiliere nun das Programm. Die Kompilierung sollte dabei mit einem Fehler enden, wobei das EiffelStudio so aussieht:



Wenn du dir die Fehlermeldung genau durchliest, dann siehst du, dass auf der zweiten Zeile, der Grund für den Kompilierungsfehler angegeben wird.

```
Error code: VUEX(2)
Error: feature of qualified call is not available to client class.
What to do: make sure feature after dot is exported to caller.
```

Du hast gelernt, dass man, wenn man ein Feature auf ein Objekt anwendet, das einen Feature call nennt. Es gibt zwei Arten von Feature calls: QUALIFIED FEATURE CALLS und UNQUALIFIED FEATURE CALLS. Bei qualified Feature calls gibt es ein ausdrückliches Zielobjekt. Bei unqualified Feature calls wird das Zielobjekt nicht ausdrücklich genannt. Die einzige Art von unqualified Feature calls sind Selbstbefehle.

Definition Qualified Feature call
Bei einem Qualified Feature call ist das Zielobjekt explizit genannt und es wird die Punkt-Notation verwendet.

Definition Unqualified Feature call
Bei einem Unqualified Feature call wird das Zielobjekt nicht explizit genannt. Unqualified Feature calls sind immer Selbstbefehle.

Wissenssicherung 3.4

Welche der folgenden Feature calls sind qualified? Welche sind unqualified?

1. ***Current***.*play*
2. *load_music*
3. *player*.*load_music*

Um einem Objekt (auch sich selbst) einen Befehl zu geben, muss das Feature in der Klasse vorhanden sein. In Eiffel kann man das Vorhandensein von Features noch weiter einschränken, nämlich mit dem Exportstatus. Im Allgemeinen unterscheidet man zwischen privaten Features, die nur durch unqualified Feature calls aufgerufen werden

dürfen, und Features, die von allen gesehen und benutzt werden können (also auch mittels qualified Feature calls). Features mit eingeschränktem Exportstatus stehen in einer Featureabschnitt mit der Überschrift

feature $\{NONE\}$ −− Something here

Betrachte die Klasse MEMORY_GAME_SCENE in Listing 3.3.

Die Features *initialize_scene* und *music_player* kann jeder aufrufen. Die Features *switch_player*, *start_animations*, *board*, *player1* und *player2* können nur als unqualified Feature calls benutzt werden, das heisst nur innerhalb der Klasse MEMORY_GAME_SCENE selbst. Gekennzeichnet werden die eingeschränkten Features, indem sie in den Abschnitt einer Featureankündigung, die **feature**{NONE} heisst, gelegt werden. Das NONE bedeutet, dass die folgenden Features nicht exportiert werden, das heisst nur intern in der Klasse benutzt werden können. Die Featureankündigung bestimmt also den EXPORTSTATUS der nachfolgenden Features.

Definition Exportstatus
Der Exportstatus eines Features bestimmt, ob es von anderen Klassen aus aufgerufen werden kann.

Wissenssicherung 3.5

Schau dir die folgende Klasse aus Listing 3.4 an und entscheide für jedes Feature

1. *ob es mittels qualified Feature call aufgerufen werden kann.*

2. *ob es eine Query oder ein Command ist.*

3.4 Attribute und Funktionen

Features lassen sich in Queries und Commands aufteilen. Das weisst du nun schon. Schauen wir uns die Queries nochmals genauer an. In der Definition von Query heisst es, dass Queries es “erlauben einen Aspekt des Zustands des Objekts abzufragen”.

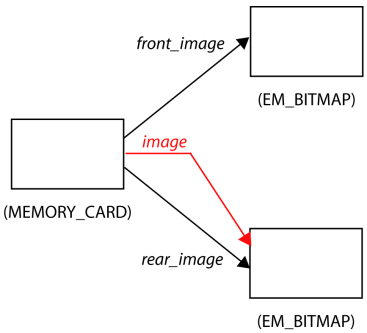
Sehen wir uns die Klasse MEMORY_CARD (siehe Listing 3.5) und insbesondere die Query *image* nochmals genauer an. Über die Query *image* bekommen wir Zugriff auf das Bild, das für das Memorykartenobjekt, im Moment gerade dargestellt ist. Wenn die Karte verdeckt ist, dann sollten wir über *image* das Bild der Rückseite (*rear_image*) erhalten; wenn sie aufgedeckt wurde, dann sollte es die Vorderseite (*front_image*) sein. Aufgezeichnet sieht das ganze so aus:

Listing 3.3: Klasse MEMORY_GAME_SCENE

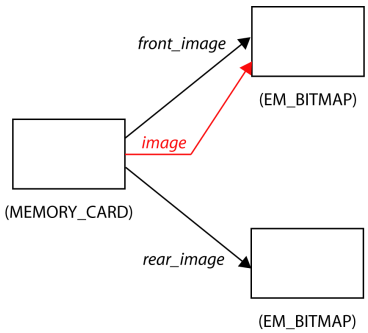
```
class

    MEMORY_GAME_SCENE

feature -- Initialization

    initialize_scene is
            -- Create all needed objects and put them into 'main_container'.
        do
            -- Something here
            start_animations
        end

feature -- Access

    music_player: MEMORY_MUSIC_PLAYER
        -- Player for background music

feature {NONE} -- Implementation commands

    switch_player is
            -- Switch between 'player1' and 'player2'.
        do
            -- Something here
        end

    start_animations is
            -- Start animating all EM_SPRITES in main_container.
        do
            -- Something here
        end

feature {NONE} -- Implementation queries

    board: MEMORY_BOARD
        -- Board on which the game is played

    player1: MEMORY_PLAYER
        -- First player

    player2: MEMORY_PLAYER
        -- Second player

end
```

Listing 3.4: Klasse MEMORY_MENU_SCENE

```
class

    MEMORY_MENU_SCENE

feature -- Initialization

    initialize_scene is
            -- Initialize the scene
        local
            elements: MEMORY_GRAPHICAL_ELEMENTS
        do
        end

feature -- Access

    selected_nr : INTEGER
        -- The number of the selected entry

feature {NONE} -- Utilities

    update_menu_entries is
            -- Updates the 'menu_entry'
        do
        end

    add_menu_entries_to_scene is
            -- Add menu entries to 'scene'.
        do
        end

feature {NONE} -- Implementation

    menu_entries: HASH_TABLE [EM_STRING, INTEGER]
        -- The menu entries

    menu_entry_scenes: HASH_TABLE [EM_SCENE, INTEGER]
        -- The menu entry scenes

end
```



(a) Karte nicht aufgedeckt



(b) Karte aufgedeckt

Nehmen wir nun mal an, dass ein anderes Objekt, zum Beispiel das MEMORY_BOARD für eine bestimmte Karte abfragen möchte, ob die Karte aufgedeckt ist. Dann müsste die Klasse MEMORY_CARD natürlich eine Query zur Verfügung stellen, die dies erlaubt. Das einfachste wäre, wenn einfach geschaut wird, ob die Query *image* dasselbe Objekt zurückgibt, wie die Query *front_image*. Wenn das der Fall ist, dann ist die Karte aufgedeckt, sonst nicht. Nun kann man diese neue Query (wir nennen sie *is_uncovered*), durch Berechnung erzeugen. Betrachte dazu die Zeilen 22 bis 30.

Den Featurebody musst du nicht verstehen. Beachte jedoch, dass dies eine Query ist, obwohl ein Featurebody (Zeilen 25-28) da ist. Es gibt also zwei Arten von Queries. Die erste Art sieht im Allgemeinen so aus (die kennst du schon aus Abschnitt 3.2):

```
query: SOME_CLASS
        -- Some comment
```

Queries dieser Art, nennt man ATTRIBUTE. Queries der zweiten Art (wie zum Beispiel *image*) nennt man FUNKTIONEN, da sie ein Resultat berechnen und Zugriff auf dieses geben. Die allgemeine Form von Funktionen sieht so aus:

```
query: SOME_CLASS is
        -- Some comment
    do
        -- Something here
    end
```

Attribute sind also einfach zu finden, Funktionen lassen sich weniger einfach von Commands unterscheiden. Am sichersten ist es, wenn du im Zweifelsfall schaust, ob direkt vor dem **is** in der ersten Zeile der Featuredeklaration, ein Doppelpunkt und ein Klassenname vorkommen (ohne Klammern rundherum).

Wissenssicherung 3.6

Betrachte den Klassentext in Listing 3.6. Welche Features sind Commands? Welche sind Attribute? Und welche sind Funktionen?

Listing 3.5: Klasse MEMORY_CARD mit Funktion is_uncovered

```
1  class MEMORY_CARD
2
3  feature -- Basic operations
4
5      flip is
6              -- Flip the card.
7          do
8              if is_uncovered then
9                  image := rear_image
10             else
11                 image := front_image
12             end
13         ensure
14             visibility_changed : old is_uncovered = not is_uncovered
15         end
16
17 feature -- Status
18
19     image: EM_BITMAP
20             -- Currently shown image (either 'front_image' or 'rear_image')
21
22     is_uncovered : BOOLEAN is
23         -- Was this memory card uncovered in the current turn?
24         do
25             if image = rear_image then
26                 Result := False
27             else
28                 Result := True
29             end
30         end
31
32 feature {NONE} -- Implementation
33
34     front_image : EM_BITMAP
35         -- The front image of the memory card
36
37     rear_image: EM_BITMAP
38         -- The front image of the memory card
39
40 end
```

Listing 3.6: Klasse MEMORY_BOARD

```
class

    MEMORY_BOARD

feature

    make_large is
            -- Create a memory board with large size
        do
            -- Something here
        end

     first_uncovered_card : MEMORY_CARD
        -- The first card uncovered in this turn

    make_small is
            -- Create a memory board with small size
        do
            -- Something here
        end

    width:INTEGER is
            -- The 'width' of ' current '
        do
            -- Something here
        end

    match is
            -- Match and flip accordingly .
        do
            -- Something here
        end

    is_match: BOOLEAN is
            -- Do ' first_uncovered_card ' and 'second_uncovered_card' match?
        do
            -- Something here
        end

    deal is
            -- Deal the cards .
        do
            -- Something here
        end

end
```

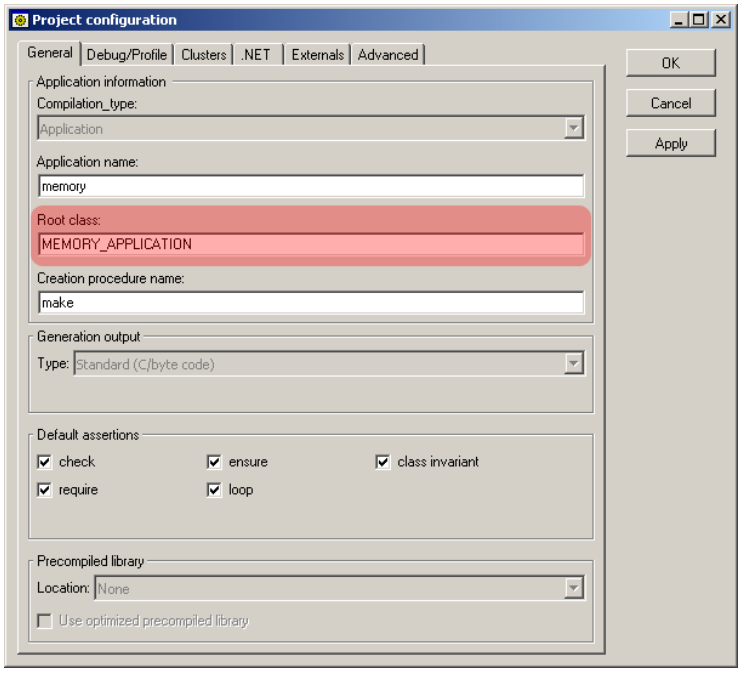


Abbildung 3.2: Dialog Projektkonfiguration

3.5 Ablauf eines Programmes und Objekterzeugung

Was passiert, wenn wir unser Programm ausführen? Die Ausführung des Programms beginnt mit der Erzeugung eines ganz bestimmten Objekts, dem sogenannten WURZELOBJEKT. Der Typ dieses Objekts wird vom Programmierer in der Ace-Datei bestimmt und kann auch im EiffelStudio bei der Projektkonfiguration nachgeschaut (und falls erwünscht verändert) werden.

Öffne dazu das Memory-Projekt im EiffelStudio und gehe zur Projektkonfiguration, indem du auf das Menu `Project ▷ Project settings` klickst.

Wie du in der Projektkonfiguration (siehe Abbildung 3.2) unter der Überschrift “Root class” erkennen kannst, ist unser Wurzelobjekt vom Typ MEMORY_APPLICATION:

Zusätzlich zur Klasse des Wurzelobjekts wird ausserdem der Name eines Features angegeben, dessen Ausführung die Ausführung des Projekts bedeutet (siehe das Feld “Root creation procedure”). In unserem Fall ist es das Feature *make*.

Das Wurzelobjekt wird (etwas salopp gesagt) durch Zauberhand erzeugt und ebenfalls durch Zauberhand wird angefangen, das Feature *make* auszuführen. Dies geschieht Zeile für Zeile. Dabei enthält eine Zeile jeweils Selbstbefehle oder Befehle an andere Objekte.

Woher kommen aber diese anderen Objekte? Auch sie müssen erzeugt werden. Dies geschieht, indem ein spezielles Schlüsselwort benutzt wird: **create**. Natürlich

muss nun auch noch gesagt werden, durch welchen Namen das neu erzeugte Objekt angesprochen werden kann (ohne einen solchen Namen, können wir ja das neue Objekte nicht verwenden). Die einfachste Form einer sogenannten Erzeugungsinstruktion (einer Zeile Code, die ein neues Objekt erzeugt), sieht folgendermassen aus:

create *objekt_name*

Beim Erzeugen von Objekten gibt es jedoch wieder Regeln, die zu befolgen sind. Die erste wurde schon angedeutet: *objekt_name* muss ein Attribut der Klasse sein, die die Erzeugung bewirkt. Beachte, dass es zwingend ein Attribut sein muss, und keine Funktion! Da *objekt_name* ein Attribut der Klasse ist, kannst du nun auch herausfinden, von welchem Typ das erzeugte Objekt sein wird - nämlich dem der in der Featuredeklaration von *objekt_name* definiert ist.

Die zweite Form erzeugt ein Objekt, auf das mittels der Query *objekt_name* zugegriffen werden kann und gibt diesem neuen Objekt sofort auch einen Befehl. Diese Form der Objekterzeugung ist für Objekte da, die irgendwelche Initialisierungen machen müssen, die nicht vergessen gehen dürfen (zum Beispiel weitere Objekte erzeugen).

create *objekt_name*. *initialisiere_dich*

Den Command *initialisiere_dich* nennt man eine ERZEUGUNGSPROZEDUR. Aber aufgepasst: Für jede Klasse ist im Klassentext genau angegeben, welche Commands als Erzeugungsprozeduren benutzt werden können. In der Klasse MEMORY_BOARD (Listing 3.7) siehst du, dass sogar zwei Erzeugungsprozeduren zur Verfügung stehen! Das bedeutet also, dass wenn ein Objekt vom Typ MEMORY_BOARD erzeugt werden soll, immer eine dieser zwei Erzeugungsprozeduren aufgerufen werden muss.

Beachte, dass die Liste der Erzeugungsprozeduren immer genau oberhalb der ersten **feature**-Klausel stehen muss. Natürlich gibt es haufenweise Klassen, die keine Erzeugungsprozeduren auflisten. Objekte eines solchen Typs müssen durch die erste Form erzeugt werden.

Wissenssicherung 3.7

Fülle den Featurebody des Features make der Klasse FICTIVE_MEMORY_CLASS[3] (siehe Listing 3.8) so aus, dass für alle Attribute der Klasse ein neues Objekt erzeugt wird. Um herauszufinden, welche Erzeugungsprozeduren du benutzen kannst, schaust du in den Klassen des Memory-Projekt nach.

3.6 Wie echte Klassen aussehen

Bis jetzt hast du nur Klassentexte gesehen, die hier im Leitprogramm vorgekommen sind. Diese wurden alle ein bisschen verändert, so dass nur Dinge vorkommen, die du verstehst. Nun wirst du einen ersten vollständigen, unveränderten Klassentext sehen.

Nun kommt er also: der unverhüllte Klassentext von Klasse MEMORY_PLAYER (siehe Listing 3.9). Nimm einen Textmarker zur Hand und versuche als erstes alle Teile des Quelltexts, die du bereits kennst, zu markieren. Gehe den Quelltext von oben

[3] FICTICIOUS_MEMORY_CLASS ist nur eine fiktive Klasse, du findest sie nicht im Memory-Projekt!

Listing 3.7: Klasse MEMORY_BOARD

```
class

    MEMORY_BOARD

create
    make_large, make_small

feature {NONE} -- Initialization

    make_large is
            -- Create a memory board of large size
        do
            ...
        end

    make_small is
            -- Create a memory board of small size
        do
            ...
        end

    ...

end
```

Zeile für Zeile durch. Falls Fragen auftauchen, schreibe sie dir auf und wende dich an einen Mitschüler, der gleich weit ist, deinen Lehrer oder deinen Tutor.

Listing 3.9: Die echte Klasse MEMORY_PLAYER

```
1
2  indexing
3      description : "Objects representing a memory player"
4
5  class
6      MEMORY_PLAYER
7
8  inherit
9      EM_DRAWABLE_CONTAINER [EM_DRAWABLE]
10         undefine copy, is_equal end
11
12     EM_SHARED_STANDARD_FONTS
13         export {NONE} all
14         undefine copy, is_equal , default_create end
15
16     EM_SHARED_BITMAP_FACTORY
```

Listing 3.8: Klasse FICTICIOUS_MEMORY_CLASS

```
class FICTICIOUS_MEMORY_CLASS

create
    make

feature

    make is
            -- Create all the needed objects.
        do
            -- To be filled by you!
        end

    elements: MEMORY_GRAPHICAL_ELEMENTS
            -- Elements for the menu scene

    game_scene: MEMORY_GAME_SCENE
            -- Game scene

    board: MEMORY_BOARD
            -- Memory board

    player: PLAYER is
            -- Player
        do
        end

end
```

```
17          export {NONE} all
18          undefine default_create end
19
20      MEMORY_FONT
21
22  create
23      make_with_name_and_color
24
25  feature {NONE} -- Initialization
26
27      make_with_name_and_color (a_name: STRING; a_color: EM_COLOR) is
28              -- Create object with 'a_name'.
29          require
30              name_not_void: a_name /= Void
31              name_not_empty: not a_name.is_empty
32               color_not_void : a_color /= Void
33          do
```

```
34              make
35              create name.make_from_string (a_name)
36              points := 0
37              color := a_color
38
39              create name_text.make (name, small_font)
40              name_text.set_x_y (x + 10, y + 10)
41              create points_text.make (points.out, small_font)
42              points_text.set_x_y (x + 10, y + 50)
43
44              create rectangle.make_from_coordinates (0, 0, width, height)
45              rectangle.set_line_color (color)
46              rectangle.set_line_width (4)
47              rectangle.set_filled (false)
48          ensure
49              name_set: name.is_equal (a_name)
50              color_set: color = a_color
51          end
52
53
54  feature -- Basic operations
55
56      show is
57              -- Show the player.
58          do
59              extend (name_text)
60              extend (points_text)
61              rectangle.set_size (width, height)
62              extend (rectangle)
63          end
64
65      increase_points is
66              -- Increase 'points' by one.
67          do
68              points := points + 1
69              points_text.set_value (points.out)
70          ensure
71              points_increased: points = old points + 1
72          end
73
74      reset is
75              -- Reset score for new game.
76          do
77              points := 0
78              points_text.set_value (points.out)
79          ensure
80              points_zero: points = 0
81          end
82
83  feature -- Access
```

```
84
85      name: STRING
86          -- Name of the player
87
88      points : INTEGER
89          -- Current number of points of the player
90
91      color : EM_COLOR
92          -- Color of player
93
94  feature {NONE} -- Implementation
95
96       points_text : EM_STRING
97          -- Text showing current score
98
99      name_text : EM_STRING
100         -- Text showing name of player
101
102     rectangle : EM_RECTANGLE
103         -- Border around player info
104
105 invariant
106
107     name_not_void: name /= Void
108     name_not_empty: not name.is_empty
109      points_positive_or_zero : points >= 0
110
111 end
```

Nachdem du das getan hast, findest du in Listing 3.10 die Lösung mit einigen zusätzlichen Bemerkungen.

8 - 20: Hier werden alle Klassen aufgelistet, von denen MEMORY_PLAYER “erbt”. Vererbung wird im Abschnitt 4.2 erklärt.

27: Auf dieser Zeile findest du eine weitere Art von Feature: ein Feature mit einem sogenannten Argument. Features mit Argumenten werden im Abschnitt 4.1 im nächsten Kapitel behandelt.

29-32: Hier findest du die sogenannten Vorbedingungen für ein Feature. Diese werden in diesem Leitprogramm nicht behandelt.

35-47, 59-62, 68-69: Jeder dieser Blöcke beinhaltet viele Featurecalls, sowie einige spezielle Konstrukte. Diese Zeilen sind unwichtig für dich. Einige Zeilen wirst du nach Durcharbeitung des nächsten Kapitels verstehen, andere werden jedoch in diesem Leitprogramm nicht behandelt.

48-50, 70-71: Hier findest du die sogenannten Nachbedingungen für ein Feature. Diese werden in diesem Leitprogramm nicht behandelt.

105-109: Hier findest du wiederum ein Element, das nicht in diesem Leitprogramm behandelt wird: die Invariante.

Listing 3.10: Die echte Klasse MEMORY_PLAYER mit angestrichenen Teilen

```
1
2  indexing
3      description : "Objects␣representing␣a␣memory␣player"
4
5  class
6      MEMORY_PLAYER
7
8  inherit
9      EM_DRAWABLE_CONTAINER [EM_DRAWABLE]
10         undefine copy, is_equal end
11
12     EM_SHARED_STANDARD_FONTS
13         export {NONE} all
14         undefine copy, is_equal , default_create end
15
16     EM_SHARED_BITMAP_FACTORY
17         export {NONE} all
18         undefine default_create end
19
20     MEMORY_FONT
21
22 create
23     make_with_name_and_color
24
25 feature {NONE} -- Initialization
26
27     make_with_name_and_color (a_name: STRING; a_color: EM_COLOR) is
28             -- Create object with 'a_name'.
29         require
30             name_not_void: a_name /= Void
31             name_not_empty: not a_name.is_empty
32             color_not_void : a_color /= Void
33         do
34             make
35             create name.make_from_string (a_name)
36             points := 0
37             color := a_color
38
39             create name_text.make (name, small_font )
40             name_text. set_x_y (x + 10, y + 10)
41             create points_text .make (points.out, small_font )
42             points_text . set_x_y (x + 10, y + 50)
43
44             create rectangle .make_from_coordinates (0, 0, width, height)
45             rectangle . set_line_color (color)
46             rectangle . set_line_width (4)
47             rectangle . set_filled ( false )
48         ensure
```

```
49                name_set: name.is_equal (a_name)
50                color_set : color = a_color
51            end
52
53
54    feature -- Basic operations
55
56        show is
57                -- Show the player.
58            do
59                extend (name_text)
60                extend ( points_text )
61                rectangle . set_size (width, height)
62                extend (rectangle)
63            end
64
65        increase_points is
66                -- Increase ' points ' by one.
67            do
68                points := points + 1
69                points_text . set_value (points.out)
70            ensure
71                points_increased : points = old points + 1
72            end
73
74        reset is
75                -- Reset score for new game.
76            do
77                points := 0
78                points_text . set_value (points.out)
79            ensure
80                points_zero : points = 0
81            end
82
83    feature -- Access
84
85        name: STRING
86            -- Name of the player
87
88        points: INTEGER
89            -- Current number of points of the player
90
91        color: EM_COLOR
92            -- Color of player
93
94    feature {NONE} -- Implementation
95
96        points_text : EM_STRING
97            -- Text showing current score
98
```

```
99      name_text: EM_STRING
100         -- Text showing name of player
101
102     rectangle : EM_RECTANGLE
103         -- Border around player  info
104
105 invariant
106
107     name_not_void: name /= Void
108     name_not_empty: not name.is_empty
109       points_positive_or_zero  :  points  >= 0
110
111 end
```

3.7 Check-Up

Als Check-Up für dieses Kapitel sollst du die Features *new_large_game* und *new_small_game* in der Klasse MEMORY_GAME_SCENE implementieren. Das Feature *new_large_game* wird aufgerufen, sobald der Benutzer die Taste 'l' drückt; das Feature *new_small_game* sobald er 's' drückt.

Beide Features geben dem Benutzer die Möglichkeit ein laufendes Spiel abzubrechen und entweder auf einem grossen oder einem kleinen Spielbrett neu zu beginnen. Um die Features zu implementieren, gehst du folgendermassen vor:

1. Gehe zum Feature *new_large_game* in der Klasse MEMORY_GAME_SCENE und füge eine neue Zeile ein. Es sollte nun so aussehen und deine Eingabe sollte auf der leeren Zeile beginnen:

```
new_large_game is
        -- Reset board ( large ).
    do
        ----------------------------
        -- FILLED IN BY STUDENTS --

        ----------------------------
    end
```

2. Zuerst musst du das Spielbrett zurücksetzen. Versuche herauszufinden, welchen Typ das bestehende Spielbrett hat, indem du die Features der Klasse MEMORY_GAME_SCENE genau anschaust.

3. Nun schaust du dir die Klasse an, welche das Memorybrettobjekt beschreibt, und versuchst herauszufinden, welchen Befehl du ihm geben musst. Beachte, dass du kein neues Spielbrett-Objekt erzeugen, sondern nur das bestehende zurücksetzen musst, indem du den passenden Befehl ausführen lässt.

4. Gehe zum Feature *new_large_game* und tippe nun den Befehl an das richtige Zielobjekt ein. Kompiliere. Falls ein Kompilierungsfehler auftritt, verbesserst du ihn, bis das Programm kompiliert.

5. Nun musst du natürlich auch den Punktestand der beiden Spieler zurücksetzen. Gehe auf eine neue Linie und gehe gleich vor, wie bei der ersten Zeile. Vergiss wieder nicht zu kompilieren
6. Lasse dein verändertes Programm laufen und überprüfe, ob es das gewünschte tut.
7. Implementiere nun auch das zweite Feature *new_small_game*. Der einzige Unterschied ist, dass mit diesem Feature ein kleines Spielbrett angezeigt wird. Auch hier musst du kein neues Spielbrett-Objekt erzeugen!

3.8 Antworten zu Wissenssicherungen

Lösung Wissenssicherung 3.1

1. Eine Klasse wird benötigt[4].
2. Null bis unendlich viele Objekte können von einer Klasse beschrieben werden.

Lösung Wissenssicherung 3.2

Beispiel: Eine Klasse ist wie ein Bauplan für Häuser, ein Objekt ist wie ein konkretes, auf einem Grundstück stehendes Haus. Lass dir von einem Kollegen Feedback zu deiner Analogie geben.

Lösung Wissenssicherung 3.3

1. Klasse MEMORY_PLAYER
2. Features *show*, *name*, *points*, *increase_points*, *reset* und *color*
3. Klassen MEMORY_PLAYER, STRING, INTEGER, EM_COLOR
4. *name*, *points*, *color*
5. *show*, *increase_points*, *show_player_image*
6. Zeilen 20 bis 22
7. Feature *points* hat keinen Featurebody
8. Auf den Zeilen 7, 9, 13, 16, 19, 21, 25, 27, 31

Lösung Wissenssicherung 3.4

[4] Fortgeschrittene objekt-orientierte Konzepte wie Vererbung werden hier nicht berücksichtigt.

1. Qualified Feature call
2. Unqualified Feature call
3. Qualified Feature call

Lösung Wissenssicherung 3.5

1. Features, die mittels qualified Feature call aufgerufen werden können: *initialize_scene*, *selected_nr*
2. Queries: *selected_nr*, *menu_entries*, *menu_entry_scenes*; Commands: *initialize_scene*, *update_menu_entries*, *add_menu_entries_to_scene*

Lösung Wissenssicherung 3.6

- board
- player1
- player2
- current_player
- music_player

Lösung Wissenssicherung 3.7

- Commands: *make_large, make_small, match, deal*
- Attribute: *first_uncovered_card*
- Funktionen: *width, is_match*

Lösung Wissenssicherung 3.8

```
make is
        -- Create all the needed objects.
    do
        create elements
        create game_scene.make_scene
        create board.make_small -- oder
        create board.make_large
    end
```

Kapitel 4

Featureaufrufe

Till G. Bay

Übersicht

Was lernst du hier?

Wie du im letzten Kapitel gesehen hast, geht es bei der objekt-orientierten Programmierung um Instanzen von Klassen, den Objekten und den Befehlen, die man auf diesen Objekten ausführen kann. Das Zusammenspiel der Objekte wird also durch die Abfolge der Ausführungen dieser Befehle bestimmt. Wie du gesehen hast, nennt man die Befehle Commands.

Du hast aber auch gehört, was eine Query ist und dass die Queries und Commands zusammen die Features eines Objektes ergeben. In diesem Kapitel lernst du wie man nicht nur Commands auf einem bekannten Objekt ausführen kann, sondern auch wie man mit Hilfe der Objekte, die ein bekanntes Objekt kennt, Commands auf diesen anderen Objekten ausführen kann. Das kannst du Dir so vorstellen: Du kennst natürlich alle Mitglieder deiner Familie. Dein Bruder oder deine Schwester haben aber nicht genau denselben Freundeskreis wie du. Jetzt ist es aber trotzdem möglich, dass du über dein Geschwister eine Musik-CD von einem seiner Freunde ausleihen kannst. Du führst also einen Befehl auf einem Objekt aus, das du selbst gar nicht kennst. Das klappt aber trotzdem, weil ein Objekt das du kennst (dein Geschwister) das andere Objekt kennt, und an deiner Stelle den Befehl auf ihm ausführt.

Weiter wirst du selbst neue Features in eine Klasse einfügen, und diese so erweitern. Und du wirst eine sehr mächtige Art der Wiederverwendung von Features kennen lernen.

Was tust du?

Zuerst liest du den Einführungstext über Featureaufrufe. Danach führen wir für die verschiedenen Arten von Featureaufrufen jeweils eine formale Notation ein, die die verschiedenen Arten des Featureaufrufs genau beschreiben. In den Wissenssicherungen wirst du jede Art gleich im Memory-Spiel anwenden können. Ebenfalls im Memory-

Spiel wirst du lernen, wie man für ein Zielobjekt alle seine sichtbaren Features herausfinden kann. Abschliessen wirst du das Kapitel, indem du der Klasse MEMORY_GAME_SCENE ein neues Feature hinzufügst und dieses testest.

Lernziele

Nachdem du dieses Kapitel durchgearbeitet hast,

- kannst du erklären, wie man Argumente beim Aufrufen von Features verwendet
- kannst du für ein beliebiges Zielobjekt herausfinden, welche Features (Commands und Queries) aufgerufen werden können
- kannst Aufrufesequenzen verstehen und jedem Teil einer Aufrufsequenz einen Typ zuordnen

4.1 Feature Aufrufe mit Argumenten

Wie du im Kapitel 2 erfahren hast, sieht die allgemeine Form eines Befehls an ein Zielobjekt so aus:

```
ziel_objekt_name . tu_etwas
```

Im selben Kapitel hast du auch kurze Definitionen von Commands, Queries und Features erfahren. Die vollständigen Definitionen der drei Begriffe sind hier nochmals aufgeführt (Wobei die Definition bereits in Kapitel 2 vollständig war).

Definition Feature
Die Features eines Objekts sind alle seine bekannten Queries und die anwendbaren Commands. Die Features eines Objekts bestimmen sein Verhalten und seine Merkmale.

Definition Query
Eine Query bietet Zugriff auf bekannte Objekte und ermöglicht einen Aspekt des Zustandes eines Objekts abzufragen. Queries zeichnen sich dadurch aus, dass sie den Zustand eines Objekts nicht verändern. Queries haben also immer einen Rückgabewert.

In dieser vollständigen Definition einer Query wird vom Rückgabewert gesprochen. Der Rückgabewert ist der Wert, der den Zustand des Objekts beschreibt. Der Rückgabewert hat jeweils einen Typ. Für die Klasse MEMORY_PLAYER haben zum Beispiel die drei Queries *name*, *points* und *color* die Typen STRING, INTEGER und EM_COLOR. Das bedeutet, dass man einen Wert vom Typ STRING erhält, wenn auf einem Memory-Spieler Objekt die Query *name* aufruft.

Queries werden unterteilt in Attribute und Funktionen. Attribute enthalten einen vorgegebenen Wert, den sie zurückgeben, wenn sie aufgerufen werden. Funktionen hingegen enthalten eine Berechnung eines Wertes, den sie zurückgeben, wenn sie aufgerufen werden. Der Rückgabewert von Funktionen wird zwischen dem **do** und dem **end** in einer vordefinierten Variabel gespeichert. Diese Variabel heisst sinngemäss **Result**. Das Zwischenspeichern eines Wertes in der vordefinierten Variablen **Result** erfolgt über eine Zuweisung. Zuweisungen haben folgendes Format:

```
variabel := neuer_wert
```

Auch bei Zuweisungen gilt dass, die beiden Seiten der Zuweisung Werte desselben Typs enthalten müssen. Ein Beispiel für eine Zuweisung findest Du weiter unten im Listing . Hier siehst Du, dass die Variabel *points* den neuen Wert *points* + 1 zugewiesen erhält.

Definition Command
Die Menge der Commands eines Objekts bestimmt, wie sein Zustand verändert werden kann. Nur mit Commands wir der Zustand eines Objekts verändert. Commands haben also nie einen Rückgabewert.

Der Command *increase_points* ist in der Klasse MEMORY_PLAYER wie folgt definiert:

```
1 \label{incpts}
2 increase_points is
3         -- Increase 'points' by one.
4     do
5         points := points + 1
6         points_text.set_value (points.out)
7     end
```

Wir wollen diesen Command oder eben auch dieses Feature einmal Zeile für Zeile durchsehen, um zu sehen wie man erkennen kann, dass der Command eben den Zustand eines Memoryspieler-Objekts verändert. Auf der ersten Zeile sehen wir einfach den Namen des Features oder des Commands: *increase_points* - danach folgt der Kommentar auf der zweiten Zeile, der in Worten beschreibt, was das Feature tun soll. Auf der vierten Zeile sehen wir dann wie der Zustand des Objekts verändert wird. Die Query *points* enthält den Punkte Wert, der vom Spieler bisher erreicht wurde. Jetzt wird der Wert um einen Punkt erhöht. Das geschieht mit einer Zuweisung. Der neue Wert von *points* ist gleich dem alten Wert von *points* plus eins. Wenn man auf einem Memoryspieler-Objekt mehrmals hintereinander die Query *points* aufruft, erhält man immer denselben Wert. Der Zustand wird also nicht verändert. Ruft man hingegen dazwischen *increase_points* auf, wird man nach dem Aufruf einen um eins höheren Wert von *points* zurückerhalten. Der Zustand hat sich also verändert.

Auf der fünften Zeile sehen wir wiederum einen Command. Das Format dieses Commands werden wir nun genauer ansehen.

4.1.1 Commands mit Argumenten

Als erstes fallen einem an dem Befehl auf Zeile fünf die Klammern auf, die nach dem Befehl stehen. Zwischen diesen Klammern steht ein so genanntes Argument. Ein Argument ist ein Wert, den man dem Befehl mitgibt. Der Befehl verwendet das Argument zur Ausführung. Das Format eines Befehls mit Argumenten ist das folgende:

```
ziel_objekt_name . tu_etwas ( ein_wert )
```

Die Klammern dienen dazu, dem Befehl die Argumente zu übergeben. Es können auch mehrere Argumente übergeben werden. Wenn man mehre Argumente übergibt, trennt man die einzelnen Werte mit Kommas. Das Format sieht dann so aus:

```
ziel_objekt_name . tu_etwas ( erster_wert , zweiter_wert , ...)
```

Wenn du in einer Klasse einen Command hinzufügen willst, der ein Argument erhält, musst du auch ein bestimmtest Format respektieren. Du musst dem Argument einen Namen geben und den Typ des Arguments spezifizieren. Wie du siehst, ist das Format der Deklaration eines Commands sehr ähnlich wie sein Aufruf. Hier die genaue Beschreibung:

```
ziel_objekt_name . tu_etwas ( ein_wert : EIN_TYP)
```

Wie schon beim Aufruf erwähnt, ist es auch möglich mehrere Werte als Argumente zu übergeben. Die Deklaration eines Features im Code, das mehrere Argumente erhält, erfolgt wiederum durch Auflistung der Argumente, wobei man sie nicht wie beim Aufruf mit Kommas, sondern mit Semikolons voneinander trennt. Natürlich muss wieder bei jedem Wert der Typ angegeben sein.

```
ziel_objekt_name . tu_etwas ( erster_wert : EIN_TYP;
                              zweiter_wert : EIN_TYP; ...)
```

Wissenssicherung 4.1

Schreib das Feature increase_points so um dass es die Punktzahl um die erhöht werden soll als Argument erhält.

Wenn du den Befehl jetzt noch etwas genauer anschaust, fallen Dir nicht nur die Klammern auf. Zwischen den Klammern steht nicht ein Wert, sondern eine ganze Query! Wie du gelernt hast, haben Queries immer einen Rückgabewert. Ein solcher Rückgabewert kann man auch als Wert (als Argument eben) an einen Command übergeben. Das Format eines Commands ist dann dieses:

```
ziel_objekt_name . tu_etwas ( eine_query )
```

Wissenssicherung 4.2

Markiere in diesem Aufruf

```
points_text . set_value (points.out)
```

folgende Teile:

- *das Zielobjekt*
- *das Argument*
- *die Query*
- *den Command*

4.1.2 Queries mit Argumenten

Wie du weiter oben bereits erfahren hast, verändern Queries den Zustand des Objekts nicht. Queries dienen nur dazu den Zustand eines Objekts abzufragen. Das bedeutet aber nicht, dass Queries keine Argumente erhalten können. Es ist sehr gut möglich, dass eine Query erst mit Hilfe eines Arguments, das sie erhält bestimmen kann welchen Teil des Zustandes abgefragt werden soll. Das Format einer Query mit einem Argument ist hier gezeigt:

```
ziel_objekt_name . eine_query (ein_wert)
```

Mit der Deklaration im Quelltext verhält es sich wieder wie mit den Commands, man muss noch den Typen des Wertes spezifizieren. Nun wollen wir einmal im Memory-Spiel nachsehen, wie eine solche Query mit einem Argument aussieht. Öffne dazu die Klasse MEMORY_GAME_SCENE und schau das Feature *number_of_points* an. Wie du siehst, hat dieses Feature als Argument einen Wert vom Typ MEMORY_PLAYER. Das Feature liefert also den aktuellen Punktestand des Players, den es als Argument erhält. (Bitte ignoriere für den Augenblick die Zeile mit dem **require** und die darauf folgende Zeile — es wird später erklärt werden was diese bedeuten, hier sind sie nur der Vollständigkeit wegen aufgeführt.)

```
1 number_of_points (a_player: MEMORY_PLAYER): INTEGER is
2         -- Number of points ‘ a_player ’ has
3     require
4         a_player_not_void: a_player /= Void
5     do
6         Result := a_player.points
7     end
```

Genau wie bei den Commands ist es auch bei den Queries mit Argumenten möglich als Argument wiederum eine Query mitzugeben. Das Format der Query ändert sich dadurch nur unwesentlich, wie du unten siehst:

```
ziel_objekt_name . eine_query ( ein_weitere_query )
```

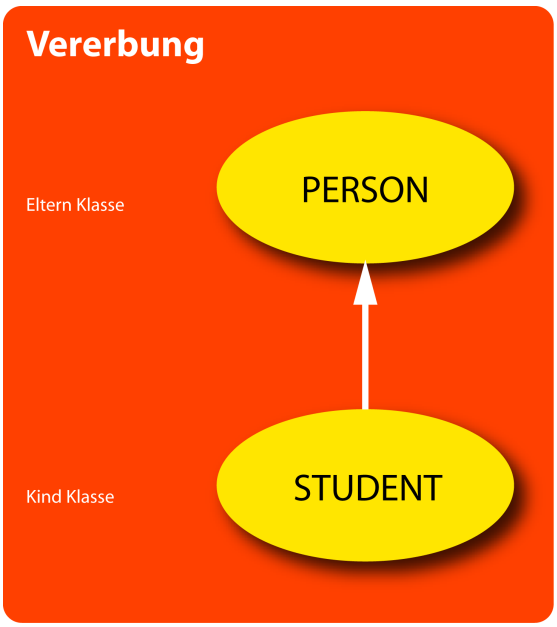


Abbildung 4.1: Vererbung

4.2 Alle Features eines Objektes

Jetzt weisst du wie alle Commands und Queries, die man auf einem Objekt aufrufen kann, aussehen. Damit kannst du jetzt beliebige Objekte verwenden und Argumente an Commands oder an Queries übergeben. Dazu musst du einfach wissen welche Commands und welche Queries ein Objekt kennt. Um das herauszufinden hast du bisher einfach die Klasse des Objekts im Editor geöffnet und hast angesehen welche Features im Quelltext der Klasse deklariert waren. Das sind aber nicht alle Features die eine Klassen kennt! In der objekt-orientierten Programmierung gibt es einen Mechanismus, der es einer Klasse erlaubt die Features einer anderen Klasse ebenfalls anzubieten.

Vererbung

Dieser Mechanismus heisst Vererbung. Wie der Name schon sagt geht es darum, dass man Fähigkeiten (eben die Features) von anderen Klassen erben kann. Genau wie man bei den Menschen oft erkennen kann, dass ein Kind seinen Eltern ähnlich sieht, ist es bei den Klassen, die voneinander erben auch. Die Features der Eltern Klasse werden dabei an die Kind Klasse weitergegeben. Zum Mechanismus der Vererbung gehören noch eine ganze Menge anderer Konzepte und viele Regeln, aber im Rahmen dieses Leitprogramms werden wir nur die einfachste Form von Vererbung ansehen. Das ist diejenige wo die Kind Klasse alle sichtbaren Features der Eltern Klasse erbt. In der Graphik 4.1 ist dargestellt wie man eine Vererbungsbeziehung darstellt.

Die Klasse PERSON enthält die Features *firstname*, *familiyname* und *age*. Die Klasse STUDENT enthält nur das Feature *legi_number*, weil die Klasse STUDENT aber von PERSON erbt (dargestellt durch den Pfeil von STUDENT nach PERSON), enthält STUDENT die drei Features von PERSON ebenfalls. In der Deklaration von STUDENT wird die Vererbungsbeziehung mit dem Schlüsselwort inherit (engl. erben) angegeben:

```
class

    KLASSEN_NAME

inherit

    NAME_DER_KLASSE_VON_DER_MAN_ERBT
```

Die beiden Klassen sehen also wie in den Listings 4.2 und 4.2 aus:

```
1  class
2
3      PERSON
4
5  feature --Access
6
7      firstname : STRING
8          -- Firstaname of a PERSON
9
10     familiyname: STRING
11         -- Lastaname of a PERSON
12
13     age: INTEGER
14         -- Age of a PERSON
15
16 end
```

```
1  class
2
3      STUDENT
4
5  inherit
6
7      PERSON
8
9  feature --Access
10
11     legi_number: INTEGER
12         -- Legi number of a STUDENT
13
14 end
```

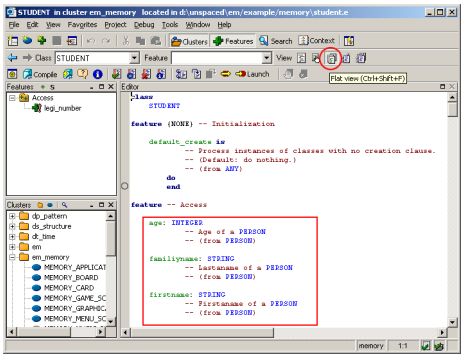


Abbildung 4.2: Flat View (Ctrl+Shift+F)

Jetzt weisst du, dass eine Klasse, in der oben eine Vererbungsbeziehung deklariert wurde, meistens noch mehr Features kennt, als diejenigen, die in ihrem Quelltext deklariert sind. Damit du herausfinden kannst, welche Features in einer Klasse alle bekannt sind, kannst du jetzt einfach die Features in den Klassen, von der sie erbt, auch noch ansehen gehen, bis du in einer Klasse bist, die von keiner anderen Klasse erbt. Dieses Vorgehen ist aber sehr umständlich, zeit intensiv und fehleranfällig. Man kann sich kaum alle Features merken, die man so findet, und weil eine Klasse auch noch von mehreren Klassen erben kann, ergeben sich oft sehr viele Features.

An dieser Stelle wird jetzt ersichtlich wie eine mächtige Entwicklungsumgebung helfen kann. EiffelStudio kann diese Arbeit abnehmen. Das gilt auch für andere Entwicklungsumgebungen für andere Programmiersprachen. EiffelStudio kennt dafür zwei Funktionen:

- Flat View
- Content Assist

Wie der Name der Flat View schon sagt, ist dies die flache Ansicht einer Klasse im Editor Fenster. In dieser Ansicht wird die ganze Vererbungshierarchie flach gepresst und aufgelöst. Man sieht also alle Features, die eine Klasse kennt. Für die Klasse STUDENT würde die Flat View wie in Graphik 4.2 gezeigt aussehen. Die Flat View wird mit dem in Graphik 4.2 markierten Button aktiviert. Die Flat View verwendet man häufig wenn man sich beim Lesen einer Klasse ein Bild machen will, was eine Klasse alles kann. In der markierten Box siehst du die drei Features, die STUDENT von PERSON erbt. Im Kommentar dieser Features wird auch angezeigt, dass diese eben von PERSON stammen. Zurück in die normale Ansicht kannst du mit einem Klick auf den Button ganz links in der Reihe der Ansichts Buttons oder mit dem Kürzel Crtl+Shift+T (diese Ansicht nennt man basic text view).

Das Content Assist hingegen ist eine Hilfe beim Tippen. Wenn man beim Tippen im Quelltext nicht mehr weiss, welche Features auf dem aktuellen Zielobjekt aufgerufen werden können, kann man mit dem Tastaturkürzel *Ctrl-Space* eine Liste der verfügbaren Features aktivieren. Das sieht dann wie in Graphik 4.3 aus. Wie du siehst, ist in dieser Liste auch gleich der Rückgabewert von Queries und auch alle Argumente und ihre Typen der Features angegeben.

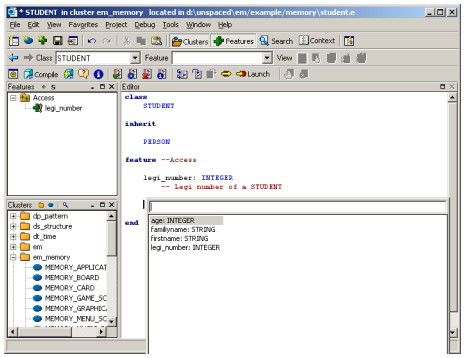


Abbildung 4.3: Content Assist (Ctrl+Space)

Sowohl in der Flat View wie auch mit dem Content Assist sieht man oft Features, von denen man nicht weiss, woher sie stammen oder wie man sie verwendet. Das ist aber nicht weiter schlimm und braucht dich nicht zu kümmern.

Wissenssicherung 4.3

Öffne die Klasse MEMORY_APPLICATION und finde alle Queries, die vom Feature *make* aus erreichbar sind und einen STRING als Rückgabewert haben. Gib für jede Query auch an, ob es eine Query mit oder ohne Argument ist.

4.3 Aufrufsequenzen

Zum Abschluss dieses Kapitels wollen wir noch ansehen wie man von einem Objekt, das man kennt zu denjenigen Objekten gelangen kann, die dieses Objekt kennt und so weiter. Du hast jetzt schon ein wenig im Gefühl, wie die objekt-orientierte Programmierung funktioniert. Man manipuliert Objekte, in dem man auf ihnen Befehle ausführt. Man kann die Manipulationen abfragen mit den Queries und abhängig von den Resultaten wieder andere Manipulationen vornehmen. So holt man sich durch Aufruf einer Query ein Zielobjekt, auf dem man dann wiederum eine Query oder einen Command ausführen kann. Wenn man wieder eine Query ausführt, erhält man wiederum einen Rückgabewert und darauf kann man wiederum eine Query oder einen Command ausführen. Dieses Spiel kann man beliebig lange wiederholen und dabei entstehen Sequenzen von Queries, die immer durch den Punkt voneinander abgetrennt sind. Diese Query Sequenzen nennt man Aufrufsequenzen. Das Format einer Aufrufsequenz ist im Kasten dargestellt:

eine_query . *eine_zweite_query* . *eine_dritte_query* . ...

Wissenssicherung 4.4

Versuche mit einer Aufrufsequenz zum Rotwert der Farbe eines Spielers des Memory-Spiels zu gelangen. Tip: Die Klasse MEMORY_GAME_SCENE kennt zwei MEMORY_PLAYER Objekte, die wiederum jeweils eine Farbe *color* des Typs EM_COLOR haben. In der Klasse EM_COLOR stehen die RGB Werte der eines Objekts des Typs EM_COLOR. Man kann sie mit dem Queries *red*, *green* und *blue* auslesen.

Aufrufsequenzen kann man also dazu verwenden

4.4 Check-Up

Nun kannst du in der Klasse MEMORY_GAME_SCENE ein neues Feature deklarieren, das *play_music* heisst, und die drei letzten Zeilen des Features *initialize_scene* enthält. Anstelle dieser drei Zeilen in *initialize_scene* kannst du dann dein eigenes Feature *play_music* aufrufen. Bravo, du hast jetzt deinen ersten eigenen Command einer Klasse hinzugefügt.

4.5 Antworten zu Wissenssicherungen

Lösung Wissenssicherung 4.1

```
increase_points (an_increment: INTEGER) is
        -- Increase ‘ points ’ by ‘ an_increment ’.
    do
        points := points + an_increment
        points_text . set_value (points . out)
    end
```

Lösung Wissenssicherung 4.2

- das Zielobjekt: *points_text*
- das Argument: *points.out*
- die Query: *out*
- den Command: *set_value (points.out)*

Lösung Wissenssicherung 4.3

Query ohne Argument:

- *application_name*: STRING
- *class_name*: STRING
- *developer_exception_name*: STRING
- *exception_trace*: STRING
- *icon_filename*: STRING
- *original_class_name*: STRING
- *original_recipient_name*: STRING
- *original_tag_name*: STRING
- *recipient_name*: STRING
- *tag_name*: STRING

Query mit Argument:

- *meaning (except: INTEGER)*: STRING

Lösung Wissenssicherung 4.4

player1 . color . red

oder

player2 . color . red

Anhang 1

Kapiteltests

1.1 Kapitel 1: Aufwärmen

Lernziele

- Der Schüler kann erklären, wie es vom Programmtext zum ausführbaren Programm kommt
- Der Schüler kann ein Programmierprojekt öffnen und daraus ein ausführbares Programm erstellen
- Der Schüler kann erklären, welche Änderungen am Programmtext welche Auswirkungen hatten

1.1.1 Frage 1

Zeichne auf wie es vom geschriebenen Quelltext eines Programms zum ausführbaren Programm kommt.

Antwort 1: (K2)

Die Zeichnung sollte folgende Elemente in dieser Reihenfolge enthalten:

1. *Eiffel-Dateien*
2. *Ace-Datei*
3. Entwicklungsumgebung
4. Kompilierung
5. Ausführbares Programm

1.1.2 Frage 2

Erkläre wozu man eine Entwicklungsumgebung verwendet.

Antwort 2: (K2)

Die Antwort sollte folgende Punkte erwähnen:

- Programm Entwicklung
- Editor für Quelltext
- Ausführen des Programms

1.1.3 Frage 3

Was geschieht bei der Kompilierung?

Antwort 3: (K1)

Die Antwort sollte folgende Punkte erwähnen:

- Übersetzung von einer höheren Sprache in Maschinensprache
- Syntaxüberprüfung
- Erstellung eines ausführbaren Programms

1.1.4 Frage 4

Erkläre wie man eine Änderung am Quelltext dann im Programm sieht.

Antwort 4: (K1)

Man kann die Änderung nach Neukompilierung des Programms sehen wenn man das Programm erneut startet.

1.2 Kapitel 2: Objekte

[1]

Lernziele

- Der Schüler kann mit Hilfe einer Analogie erklären, was Objekte sind.
- Der Schüler kann Beziehungsnetze von Objekten aufzeichnen.
- Der Schüler kennt einen Weg, um vorgegebenen Objekten Befehle zu geben.

1.2.1 Frage 1

Die meisten Computer haben heutzutage ein CD-Laufwerk. Nehmen wir an, dass wir an einer Software schreiben, mit der wir Musik-CDs abspielen wollen. Damit das ganze einfacher ist, beschränken wir uns auf CDs, die genau zwei Lieder haben. In so einem Softwaresystem hätten wir sicher ein Objekt für den CD-Spieler. Das CD-Spieler-Objekt müsste dann die eingelegte CD kennen. Dieser Beziehung geben wir den Namen *inserted_cd*. Und die eingelegte CD hat zwei Queries: *track1* und *track2*, um auf die beiden Liedobjekte zugreifen zu können.

1. Zeichne das Beziehungsnetz auf.
2. Nimm nun an, dass das CD-Spieler-Objekt eine weitere Query *current_track* hat und momentan das erste Lied gespielt wird. Zeichne diese neue Beziehung ebenfalls auf.
3. Erkläre was passiert im Beziehungsnetz, wenn zum nächsten Lied gesprungen wird?
4. Nimm an, dass beide Liedobjekte folgende Befehle ausführen können: *show_track_name* und *show_length*, wobei der Name beziehungsweise die Länge des Lieds auf dem Bildschirm ausgegeben werden. Das CD-Spieler-Objekt hat den Befehl *switch*, wobei *current_track* vom aktuell gespielten Lied auf das andere springt. Das CD-Objekt kennt den Befehl *show_artist_name* und *show_album_name*, wobei der Name des Sängers beziehungsweise der Name des Albums ausgegeben wird. Nun will das CD-Spieler-Objekt mit einer Reihe von Befehlen zuerst den Namen des Sängers, dann den Namen des Albums und jeweils für jedes Lied den Namen und die Länge des Lieds ausgeben. Schreibe die Befehle in der korrekten Reihenfolge mit Hilfe der Punktnotation auf.

Antwort 1:

1. (K3) Diese Aufgabe ist gelöst, wenn die Zeichnung richtig ist.
2. (K2) Wichtig hierbei ist herauszufinden, ob der Schüler verstanden hat, dass mehrere Objekte mit einem anderen Objekt in Beziehung stehen können.

[1] Dieser Kapiteltest enthält zwei Versionen einer Aufgabe, wobei jede der vier Teilaufgaben taxonomiert ist. Diese Form unterscheidet sich von der des vorangehenden Kapitels aber nur insoweit, dass die vier kürzeren Aufgaben in einen Gesamtkontext gefasst wurden.

3. (K2) Eine Beziehung ist nicht fest an ein bestimmtes Objekt gekoppelt, deshalb kann sich der Beziehungspfeil im Laufe eines Programms verändern.

4. (K3) Als korrekt zugelassen wird, wenn die Punkt-Notation richtig verwendet wurde und sowohl ein Selbstbefehl als wie auch die Fremdbefehle möglich sind. Die Abfolge der Befehle muss nicht zwingend richtig sein.

1.2.2 Frage 2

Die meisten Computer haben heutzutage ein DVD-Laufwerk. Nehmen wir an, dass wir an einer Software schreiben, mit der wir Film-DVDs abspielen wollen. Damit das ganze einfacher ist, beschränken wir uns auf DVDs, die genau zwei Film Szenen und ein Menu haben. In so einem Softwaresystem hätten wir sicher ein Objekt für den DVD-Spieler. Das DVD-Spieler-Objekt müsste dann die eingelegte DVD kennen. Dieser Beziehung geben wir den Namen *inserted_dvd*. Und die eingelegte DVD hat drei Queries: *menu_scene*, um auf das DVD-Menu zuzugreifen, und *scene1* und *scene2*, um auf die beiden Filmszenenobjekte zugreifen zu können.

1. Zeichne das Beziehungsnetz auf.

2. Nimm nun an, dass das DVD-Spieler-Objekt eine weitere Query *current_scene* hat und momentan das DVD-Menu abgespielt wird. Zeichne diese neue Beziehung ebenfalls auf.

3. Was passiert im Beziehungsnetz, wenn zur ersten Filmszene gesprungen wird?

4. Nimm an, dass die drei Szenenobjekte folgenden Befehl ausführen können: *show_scene_name*, wobei der Name der Szene auf dem Bildschirm ausgegeben werden. Das DVD-Spieler-Objekt hat den Befehl *go_to_next*, wobei *current_scene* von der aktuell gezeigten Szene auf die nächste springt. Das DVD-Objekt kennt den Befehl *show_name* und *show_length*, wobei der Name des Films beziehungsweise seine Länge ausgegeben wird. Nun will das DVD-Spieler-Objekt mit einer Reihe von Befehlen zuerst den Namen des Films und seine Länge und jeweils für jede Szene den Namen auf dem Bildschirm ausgeben. Schreibe die Befehle in der korrekten Reihenfolge mit Hilfe der Punktnotation auf.

Antwort 1:

1. (K3) Diese Aufgabe ist gelöst, wenn die Zeichnung richtig ist.

2. (K2) Wichtig hierbei ist herauszufinden, ob der Schüler verstanden hat, dass mehrere Objekte mit einem anderen Objekt in Beziehung stehen können.

3. (K2) Eine Beziehung ist nicht fest an ein bestimmtes Objekt gekoppelt, deshalb kann sich der Beziehungspfeil im Laufe eines Programmes verändern.

4. (K3) Als korrekt zugelassen wird, wenn die Punkt-Notation richtig verwendet wurde und sowohl ein Selbstbefehl als wie auch die Fremdbefehle möglich sind. Die Abfolge der Befehle muss nicht zwingend richtig sein.

1.3 Kapitel 3: Klassen und Objekte

Lernziele

- Der Schüler kann erklären, was eine Klasse ist und kennt den Unterschied zwischen Klasse und Objekt.
- Der Schüler kennt eine Methode, um für ein beliebiges Zielobjekt herauszufinden, welche Befehle aufgerufen werden können.
- Der Schüler weiss, wie neue Objekte erzeugt werden und kann dies selbstständig tun.

1.3.1 Frage 1

1. Erkläre wieso die Analogie, die du in der Wissenssicherung 1 dieses Kapitels gefunden hast, den Unterschied zwischen Objekt und Klasse gut widerspiegelt.
2. Wo endet die Analogie, d.h. bei welchen gelernten Konzepten wie Features, Export Status, Attributen, oder Objekterzeugung unterscheidet sich deine Analogie von den Softwarekonzepten Klasse und Objekt und deren Beziehung?

Antwort 1:

1. (K4) Die Analogie sollte insbesondere die 1:n-Beziehung zwischen Klassen und Objekten widerspiegeln. Ausserdem sollte der beschreibende Charakter einer Klasse in der Analogie enthalten sein.
2. (K4) Bei vielen Analogien ist dies der Punkt, wo die Gemeinsamkeiten enden.

1.3.2 Frage 2

Markiere in der Klassenbeschreibung, die dir dein Lehrer gibt[2], und bearbeite folgende Teilaufgaben.

1. Nenne alle Klassennamen, die in der Klassenbeschreibung vorkommen.
2. Wie heisst, die Klasse für die dieser Quelltext die Beschreibung liefert?
3. Identifiziere alle Commands dieser Klasse, die du bereits lesen kannst (die Teile des Quelltext, den du noch nicht verstehst, kannst du ignorieren).
4. Identifiziere alle Attribute dieser Klasse.
5. Identifiziere alle Funktionen dieser Klasse.
6. Welche Erzeugungsprozeduren müssen benutzt werden? Gib ein Beispiel einer Erzeugungsinstruktion, die eine der Erzeugungsprozeduren benutzt.
7. Welche Befehle können nur innerhalb der Klasse aufgerufen werden?
8. Was musst du tun, um herauszufinden, welche Befehle auf einem der durch Queries ansprechbaren Objekte ausgeführt werden können?

[2] Dazu eignen sich die meisten Klassen, so zum Beispiel EM_COLOR.

Antwort 2:

1. (K2) Alle Klassen müssen genannt werden.
2. (K2) Diese Teilaufgabe muss korrekt beantwortet werden.
3. (K2) Commands, die unbekannte Elemente enthalten (z. Bsp. Argumente) dürfen weggelassen werden. Ansonsten müssen alle Commands korrekt erkennt werden.
4. (K2) Es müssen alle Attribute korrekt erkennt werden.
5. (K2) Funktionen, die unbekannte Elemente enthalten (z. Bsp. Argumente) dürfen weggelassen werden. Ansonsten müssen alle Funktionen korrekt erkennt werden.
6. (K3) Bei Erzeugungsprozeduren mit Argumenten muss die Erzeugungsinstruktion nicht korrekt sein.
7. (K2) Alle beschränkt exportiereten Features (sowohl Commands als wie auch Queries).
8. (K3) Antwort muss beinhalten, dass durch die Featuredeklaration der Typ der Query herausgefunden werden kann, und danach im jeweiligen Klassentext die anwendbaren Commands gefunden werden können.

1.4 Kapitel 4: Featureaufrufe

Lernziele

- kannst du erklären, wie man Argumente beim Aufrufen von Features verwendet
- kannst du für ein beliebiges Zielobjekt herausfinden, welche Features (Commands und Queries) aufgerufen werden können
- kannst Aufrufesequenzen verstehen und jedem Teil einer Aufrufsequenz einen Typ zuordnen

1.4.1 Frage 1

Wie viele Argumente kann eine Query und wie viele Argumente kann ein Command haben?

Antwort 1: (K1)

Weder für Queries noch für Commands gibt es eine obere Grenze was die Anzahl der Argumente betrifft. Natürlich ist es einfacher wenn die Queries und die Commands wenig Argumente haben, weil sie da viel einfacher zu verwenden sind.

1.4.2 Frage 2

Wie kommt es, dass man im Quelltext einer Klasse nicht alle Features sieht, die man in der Klasse aufrufen kann?

Antwort 2: (K1)

In der objekt-orientierten Programmierung kennt man den Mechanismus der Vererbung, mit dem man Features einer anderen Klasse erben kann. Diese geerbten Features erscheinen dann nur in der Flat View der Klasse aber nicht im Klassen Text.

1.4.3 Frage 3

Warum kann ein Command nur am Ende einer Aufrufsequenz stehen und nicht in der Mitte?

Antwort 3: (K2)

Ein Command hat keinen Rückgabewert und deshalb erhält man durch den Aufruf eines Commands auch kein neues Zielobjekt, auf dem man eine weitere Query oder einen weiteren Command aufrufen könnte.

1.4.4 Frage 4

Wozu verwenden man Aufrufsequenzen?

Antwort 4: (K1)

Aufrufsequenzen dienen dazu von einem Objekt über die Objekte die es kennt zu weiteren Objekten zu gelangen, die diesen wiederum bekannt sind.

Anhang 2

Mediothek, Multimedia

Inhalt der Mediothek

Die Mediothek muss folgende Inhalte zwingend enthalten:

- Ordner mit zusätzlichem Material
- OOSC

Die Mediothek kann weiter noch folgende Inhalte anbieten:

- Eine Ausgedruckte Version von Touch of Class [2] enthalten

Anhang 3

Material für die Studenten

Direkt verwendbares Material

EiffelStudio Leitprogramm von Rolf Bruderer

Das Leitprogramm von Rolf Bruderer kann einfach einige Male ausgedruckt und den Schülern in einem Ordner in der Mediothek zur Verfügung gestellt werden.

Material das vorbereitet werden muss

Zusatzblatt „Aufwärmen“

Das Zusatzblatt muss vorbereitet werden. Siehe dazu das Dokument *Zusatzblatt_Aufwärmen.doc*, das mit diesem Leitprogramm mitgeliefert wird.

Anhang 4

Quellen

[1] Joseph Bergin. An object-oriented bedtime story. Available online at: http://csis.pace.edu/ bergin/Java/OOStory.html.

[2] Bertrand Meyer. Touch of class - learning to program well with object technology and design by contract. Available online under: http://se.inf.ethz.ch/touch.

[3] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.

Anhang 5

Change Log

5.1 Änderungen seit der Zwischenabgabe

- Schlüsselworte werden im Ersten Kapitel vorgestellt mit einheitlichem **style**
- Papier und Computer Icons in der Arbeitsanleitung entfernt, weil das Papiericon zu wenig oft verwendet wird.
- Vorgehen-Icon wird nur noch im Abschnitt “Was tust du?” jedes Kapitels verwendet.
- Festlegung von einheitlichem Vokabular und Schreibweise.

  - objekt-orientiert
  - Memory-Spiel
  - Memoryspieler-Objekt
  - DVD-Spieler-Objekt, DVD-Spieler, DVD-Objekt
  - Musik-CD, CD-Laufwerk, CD-Spieler-Objekt, CD-Spieler, CD-Objekt
  - Quelltext (statt: Quellcode, Programmtext)
  - Memory-Programm (statt Memory-Spiel-Programm), Programmierprojekt, Programmiersprache, Programmtext, Eiffel-Programm, Computerprogramm
  - Projekt-Datei, Memory-Projekt (statt Memory-Spiel-Projekt), Eiffel-Projekt, Projekt-Ordner, Projekt-Verzeichnisse, Projektkonfiguration
  - EiffelStudio
  - Datei (statt File), Ace-Datei, Eiffel-Datei, Textdatei, Dateitypen, Zip-Datei, Bilddateien, Dateiname
  - Ordner oder Verzeichnis (statt Directory), Projekt-Ordner, Verzeichnisstruktur
  - Cluster, Cluster-Ansicht, Memory-Cluster, Feature-Ansicht
  - Eiffel-Klassen, Eiffel-Editor
  - du, dich, dein, euer, ihr (klein)
  - Menu, Menuleiste

- Festlegung von einheitlicher Formatierung.
  - Neuer Latex-Befehl *featurename* für Featurenamen
  - Neuer Latex-Befehl classname für Klassennamen
- Appendixnummerierung mit Zahlen statt mit Buchstaben
- Quellen sind richtig formatiert und am richtigen Ort
- **Result** eingeführt
- Unterscheidung von Attribut und Funktion eingeführt
- Ein Teil der Kapiteltests adressiert jetzt mehrere Lernziele gleichzeitig, dafür wurde die vorgegebene Anzahl von vier Kapiteltest zum Teil angepasst

5.2 Änderungen, die wir nicht mehr integriert haben

- Anhang mit einer Beschreibung aller Compiler-Nachrichten (Fehlermeldungen)
- Wir haben kein weiteres Material für die schnelleren Schüler integriert, weil diese einfach im Leitprogramm weiterarbeiten können. Das Leitprogramm behandelt in seiner heutigen Fassung ja erst die ersten Vier Kapitel, der von uns vorgeschlagenen neun. Sind diese Kapitel einmal geschrieben wird es ein leichtes sein auf der Basis des Memory-Projekts eine Übung auszuarbeiten in der die schnelleren Schüler das Spiel erweitern.