Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 4
Today

- A bit of logic
- Understanding contracts (preconditions, postconditions, and class invariants)
- Entities and objects
- Object creation
Propositional Logic

- Constants: True, False
- Atomic formulae (propositional variables): P, Q, ...
- Logical connectives: not, and, or, implies, =
- Formulae: φ, χ, ... are of the form
  - True
  - False
  - P
  - not φ
  - φ and χ
  - φ or χ
  - φ implies χ
  - φ = χ
Propositional Logic

Truth assignment and truth table
- Assigning a truth value to each propositional variable

Tautology
- **True** for all truth assignments
  - P or (not P)
  - not (P and (not P))
  - (P and Q) or ((not P) or (not Q))

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>P implies Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

Contradiction
- **False** for all truth assignments
  - P and (not P)
Propositional Logic

Satisfiable

- True for at least one truth assignment

Equivalent

- $\phi$ and $\chi$ are equivalent if they are satisfied under exactly the same truth assignments, or if $\phi = \chi$ is a tautology
Tautology / contradiction / satisfiable?

- $P \text{ or } Q$
  - satisfiable
- $P \text{ and } Q$
  - satisfiable
- $P \text{ or } (\neg P)$
  - tautology
- $P \text{ and } (\neg P)$
  - contradiction
- $Q \text{ implies } (P \text{ and } (\neg P))$
  - satisfiable
Does the following equivalence hold? Prove. 

\[(P \implies Q) = (\neg P \implies \neg Q)\]

**F**

Does the following equivalence hold? Prove. 

\[(P \implies Q) = (\neg Q \implies \neg P)\]

**T**

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>P implies Q</th>
<th>not P implies not Q</th>
<th>not Q implies not P</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
Useful stuff

De Morgan laws
\[ \text{not (P or Q)} = (\text{not P}) \text{ and } (\text{not Q}) \]
\[ \text{not (P and Q)} = (\text{not P}) \text{ or } (\text{not Q}) \]

Implications
\[ P \text{ implies } Q = (\text{not P}) \text{ or } Q \]
\[ P \text{ implies } Q = (\text{not Q}) \text{ implies } (\text{not P}) \]

Equality on Boolean expressions
\[ (P = Q) = (P \text{ implies } Q) \text{ and } (Q \text{ implies } P) \]
Predicate Logic

- Domain of discourse: $D$
- Variables: $x: D$
- Functions: $f: D^n \rightarrow D$
- Predicates: $P: D^n \rightarrow \{True, False\}$
- Logical connectives: not, and, or, implies, =
- Quantifiers: $\forall, \exists$
- Formulae: $\varphi, x, \ldots$ are of the form
  - $P(x, \ldots)$
  - not $\varphi | \varphi$ and $x | \varphi$ or $x | \varphi$ implies $x | \varphi = x$
  - $\forall x \varphi$
  - $\exists x \varphi$
Existential and universal quantification

There exists a human whose name is Bill Gates

\[ \exists \ h: \text{Human} \mid h.\text{name} = \text{“Bill Gates”} \]

All persons have a name

\[ \forall \ p: \text{Person} \mid p.\text{name} \neq \text{Void} \]

Some people are students

\[ \exists \ p: \text{Person} \mid p.\text{is\_student} \]

The age of any person is at least 0

\[ \forall \ p: \text{Person} \mid p.\text{age} \geq 0 \]

Nobody likes Rivella

\[ \forall \ p: \text{Person} \mid \text{not} \ p.\text{likes} (\text{Rivella}) \]

\[ \text{not} \ (\exists \ p: \text{Person} \mid p.\text{likes} (\text{Rivella})) \]
Tautology / contradiction / satisfiable?

Let the domain of discourse be INTEGER

\( x < 0 \text{ or } x \geq 0 \)

tautology

\( x > 0 \implies x > 1 \)

satisfiable

\( \forall x \mid x > 0 \implies x > 1 \)

contradiction

\( \forall x \mid x \cdot y = y \)

satisfiable

\( \exists y \mid \forall x \mid x \cdot y = y \)

tautology
Semi-strict operations

Semi-strict operators (and then, or else)

- **a and then b**
  - has same value as \(a \text{ and } b\) if \(a\) and \(b\) are defined, and has value **False** whenever \(a\) has value **False**.

\[
text /= \text{Void} \text{ and then } text.contains(\"Joe\")
\]

- **a or else b**
  - has same value as \(a \text{ or } b\) if \(a\) and \(b\) are defined, and has value **True** whenever \(a\) has value **True**.

\[
list = \text{Void} \text{ or else } list.is\_empty
\]
Strict or semi-strict?

- $a = 0$ or $b = 0$
- $a \neq 0$ and $b \neq a$ or $0$
- $a \neq \text{Void}$ and $b \neq \text{Void}$
- $a < 0$ or $\sqrt{a} > 2$
- $(a = b$ and $b \neq \text{Void})$ and not $a$.name.is_equal ("")
Assertions

**Assertion tag** (not required, but recommended)

**Condition** (required)

`balance_non_negative: balance >= 0`

Assertion clause
Property that a feature imposes on every client

\[ \text{clap}(n: \text{INTEGER}) \]

\[ \text{-- Clap } n \text{ times and update } count. \]

\[ \text{require} \]

\[ \text{not}_{\text{too}}_{\text{tired}}: count \leq 10 \]

\[ \text{n}_{\text{positive}}: n > 0 \]

A feature with no \textbf{require} clause is always applicable, as if the precondition reads

\[ \text{require} \]

\[ \text{always}_{\text{OK}}: \text{True} \]
Property that a feature guarantees on termination

\texttt{clap\,(n: INTEGER)}
-- Clap \textit{n} times and update \textit{count}.

\textbf{require}
\begin{itemize}
\item \texttt{not\_too\_tired: count <= 10}
\item \texttt{n\_positive: n > 0}
\end{itemize}

\textbf{ensure}
\texttt{count\_updated: count = old\ count + n}

A feature with no \textbf{ensure} clause always satisfies its postcondition, as if the postcondition reads

\begin{itemize}
\item \textbf{ensure}
\texttt{always\_OK: True}
\end{itemize}
Class Invariant

Property that is true of the current object at any observable point

class ACROBAT

... 

invariant

count_non_negative: count >= 0

end

A class with no invariant clause has a trivial invariant

always_OK: True
Why do we need contracts at all?

Together with tests, they are a great tool for finding bugs.

They help us to reason about an O-O program at a class- and routine-level of granularity.

They are executable specifications that evolve together with the code.

Proving (part of) programs correct without executing them is what cool people are trying to do nowadays. This is easier to achieve if the program properties are clearly specified through contracts.
Add pre- and postconditions to:

\texttt{smallest\_power (n, bound: NATURAL): NATURAL}

\begin{verbatim}
  -- Smallest x such that `n'^x is greater or equal `bound'.
  require
      ???
  do
      ...
  ensure
      ???
  end
\end{verbatim}
One possible solution

\[ \text{smallest\_power} \ (n, \ \text{bound}: \ \text{NATURAL}): \ \text{NATURAL} \]

\[ \text{-- Smallest } x \text{ such that } n^x \text{ is greater or equal } \text{`bound'}. \]

\textbf{require}

\begin{itemize}
  \item \text{n\_large\_enough}: n > 1
  \item \text{bound\_large\_enough}: \text{bound} > 1
\end{itemize}

\textbf{do}

\[ ... \]

\textbf{ensure}

\begin{itemize}
  \item \text{greater\_equal\_bound}: n ^ \text{Result} \geq \text{bound}
  \item \text{smallest}: n ^ (\text{Result} - 1) < \text{bound}
\end{itemize}

\textbf{end}
Hands-on exercise

Add invariants to classes `ACROBAT_WITH_BUDDY` and `CURMUDGEON`.

Add preconditions and postconditions to feature `make` in `ACROBAT_WITH_BUDDY`.
Class **ACROBAT_WITH_BUDDY**

class **ACROBAT_WITH_BUDDY**

inherit **ACROBAT**

redefine
twirl, clap, count
end

create
make

feature
make (p: ACROBAT)
do
-- Remember `p' being
-- the buddy.
end

clap (n: INTEGER)
do
-- Clap `n' times and
-- forward to buddy.
end

twirl (n: INTEGER)
do
-- Twirl `n' times and
-- forward to buddy.
end

count: INTEGER
do
-- Ask buddy and return his
-- answer.
end

buddy: ACROBAT
end
Class CURMUDGEON

class CURMUDGEON

inherit ACROBAT
    redefine clap, twirl end

feature
    clap (n: INTEGER)
    do
        -- Say "I refuse".
    end

    twirl (n: INTEGER)
    do
        -- Say "I refuse".
    end
end
Entity vs. object

In the class text: an entity

$joe$: \textit{STUDENT}

In memory, during execution: an object

\begin{itemize}
\item \textit{STUDENT}
\item \textit{ASSISTANT}
\item \textit{COURSE}
\item \textit{MARK}
\item \textit{PROFESSOR}
\end{itemize}
class
  INTRODUCTION_TO_PROGRAMMING
inherit
  COURSE
feature
  execute
    -- Teach `joe` programming.
    do
      -- ???
      joe.solve_all_assignments
    end
end

joe: STUDENT
  -- A first year computer science student
end
Initial state of a reference?

In an instance of `INTRODUCTION_TO_PROGRAMMING`, may we assume that `joe` is attached to an instance of `STUDENT`?

This object has been created (by someone else)

(MEMORY)

Where does this one come from?

By someone else
Initially, \textit{joe} is not attached to any object: its value is a \textit{Void} reference.
States of an entity

During execution, an entity can:

- Be attached to a certain object
- Have the value Void
States of an entity

- To denote a void reference: use \texttt{Void} keyword
- To create a new object in memory and attach \texttt{x} to it: use \texttt{create} keyword

\[
\text{create } x
\]

- To find out if \texttt{x} is void: use the expressions

\[
\begin{align*}
\texttt{x} &= \texttt{Void} \ (\text{true iff } \texttt{x} \text{ is void}) \\
\texttt{x} &\neq \texttt{Void} \ (\text{true iff } \texttt{x} \text{ is attached})
\end{align*}
\]
Those mean void references!

The basic mechanism of computation is feature call

\[ x.f(a, ...) \]

Since references may be void, \( x \) might be attached to no object

The call is erroneous in such cases!
Why do we need to create objects?

Shouldn’t we assume that a declaration

\[ \textit{joe: STUDENT} \]

creates an instance of \textit{STUDENT} and attaches it to \textit{joe}?
Those wonderful void references!

Married persons:

Unmarried person:
Those wonderful void references!

Imagine a DECK as a list of CARD objects

Last `next` reference is void to terminate the list.
Creation procedures

- Instruction **create** \( x \) will initialize all the fields of the new object attached to \( x \) with default values.

- What if we want some specific initialization? E.g., to make object consistent with its class invariant?

```plaintext
Class CUSTOMER
...
    id: STRING
invariant
    id /= Void

- Use creation procedure:
  ```
  create a_customer.set_id(“13400002“)
  ```
Class CUSTOMER
create
set_id

feature
id: STRING

-- Unique identifier for Current.

set_id(a_id: STRING)

-- Associate this customer with `a_id`.
require
  a_id_exists: a_id /= Void
id := a_id
ensure
  id_set: id = a_id

invariant
  id_exists: id /= Void
end

List one or more creation procedures

May be used as a regular command and as a creation procedure

Is established by set_id
To create an object:

- If class has no `create` clause, use basic form:
  
  ```
  create x
  ```

- If the class has a `create` clause listing one or more procedures, use
  
  ```
  create x.make(...) 
  ```

  where `make` is one of the creation procedures, and `(...)` stands for arguments if any.
Some acrobatics

class DIRECTOR
create prepare_and_play
feature
  acrobat1, acrobat2, acrobat3: ACROBAT
  friend1, friend2: ACROBAT_WITH_BUDDY
  author1: AUTHOR
  curmudgeon1: CURMUDGEON

prepare_and_play
  do
    author1.clap(4)
    friend1.twirl(2)
    curmudgeon1.clap(7)
    acrobat2.clap(curmudgeon1.count)
    acrobat3.twirl(friend2.count)
    friend1.buddy.clap(friend1.count)
    friend2.clap(2)
  end
end

What entities are used in this class?

What’s wrong with the feature prepare_and_play?
class DIRECTOR
create prepare_and_play
feature
  acrobat1, acrobat2, acrobat3: ACROBAT
  friend1, friend2: ACROBAT_WITH_BUDDY
  author1: AUTHOR
  curmudgeon1: CURMUDGEON

prepare_and_play
  do
  1  create acrobat1
  2  create acrobat2
  3  create acrobat3
  4  create friend1.make_with_buddy (acrobat1)
  5  create friend2.make_with_buddy (friend1)
  6  create author1
  7  create curmudgeon1
  end
end
Meet Teddy