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 $(\neg P)$
  - $(P \text{ and } (\neg P))$
  - $(P \text{ and } Q) \text{ or } ((\neg P) \text{ or } (\neg Q))$

Contradiction
- **False** for all truth assignments
  - $P \text{ and } (\neg P)$

<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>
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?

- \( P \lor Q \) satisfiable
- \( P \land Q \) satisfiable
- \( P \lor (\neg P) \) tautology
- \( P \land (\neg P) \) contradiction
- \( Q \implies (P \land (\neg P)) \) satisfiable
Equivalence

Does the following equivalence hold? Prove.

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

Does the following equivalence hold? Prove.

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

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>P \implies Q</th>
<th>\neg P \implies \neg Q</th>
<th>\neg Q \implies \neg 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>
De Morgan laws

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

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

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: \( \phi, x, ... \) are of the form
  - \( P(x, ...) \)
  - \( \text{not} \ \phi \ | \ \phi \ \text{and} \ x \ | \ \phi \ \text{or} \ x \ | \ \phi \ \text{implies} \ x \ | \ \phi = x \)
  - \( \forall x \ \phi \)
  - \( \exists x \ \phi \)
Existential and universal quantification

There exists a human whose name is Bill Gates
\( \exists \ h: \text{Human} \mid h.name = \text{“Bill Gates”} \)

All persons have a name
\( \forall \ p: \text{Person} \mid p.name \neq \text{Void} \)

Some people are students
\( \exists \ p: \text{Person} \mid p.is\_student \)

The age of any person is at least \( 0 \)
\( \forall \ p: \text{Person} \mid p.age \geq 0 \)

Nobody likes Rivella
\( \forall \ p: \text{Person} \mid \text{not } p.likes(\text{Rivella}) \)

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

Let the domain of discourse be INTEGER

- $x < 0$ 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 and b* if *a* and *b* are defined, and has value **False** whenever *a* has value **False**.

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

- **a or else b**
  has same value as *a or b* if *a* and *b* are defined, and has value **True** whenever *a* has value **True**.

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

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

Assertion tag (not required, but recommended)

\texttt{balance\_non\_negative}: balance \geq 0

Condition (required)

Assertion clause
Property that a feature imposes on every client

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

-- Clap \( n \) times and update \( \text{count} \).

require
\[ \begin{align*}
\text{not\_too\_tired} & : \text{count} \leq 10 \\
\text{n\_positive} & : \, n > 0
\end{align*} \]

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

require
\[ \begin{align*}
\text{always\_OK} & : \, \text{True}
\end{align*} \]
Postcondition

Property that a feature guarantees on termination

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

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

\[
\text{require}
\]

\[
\text{not\_too\_tired: count} \leq 10
\]

\[
\text{n\_positive: } n > 0
\]

\[
\text{ensure}
\]

\[
\text{count\_updated: count} = \text{old count} + n
\]

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

\[
\text{ensure}
\]

\[
\text{always\_OK: True}
\]
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
Add pre- and postconditions to:

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

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

require
  
  
  do
  ...
  
  ensure

end
One possible solution

```plaintext
smallest_power (n, bound: NATURAL): NATURAL
    -- Smallest x such that `n'^x is greater or equal `bound'.

require
    n_large_enough: n > 1
    bound_large_enough: bound > 1

ensure
    greater_equal_bound: n ^ Result >= bound
    smallest: n ^ (Result - 1) < bound

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

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

```plaintext
class ACROBAT_WITH_BUDDY
  inherit ACROBAT
  redefine twirl, clap, count
end

create
  make
end

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

\[\text{joe: STUDENT}\]

In memory, during execution: an object
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
In an instance of `INTRODUCTION_TO_PROGRAMMING`, may we assume that `joe` is attached to an instance of `STUDENT`?
By default

Initially, \textit{joe} is not attached to any object: its value is a \textbf{Void} reference.

\begin{tikzpicture}
  \node (joe) [draw] {joe};
  \node (void) [draw, right of=joe] {Void reference};
  \draw [->] (joe) -- (void);
\end{tikzpicture}
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 `Void` keyword
- To create a new object in memory and attach $x$ to it: use `create` keyword

\[
\text{create } x
\]

- To find out if $x$ is void: use the expressions

\[
\begin{align*}
x &= \textbf{Void} \ (\text{true iff } x \text{ is void}) \\
x &\neq \textbf{Void} \ (\text{true iff } x \text{ is attached})
\end{align*}
\]
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

\[ j\text{oe}: \text{STUDENT} \]

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

Married persons:

\[(\text{PERSON}) \rightarrow \text{spouse} \rightarrow (\text{PERSON})\]

Unmarried person:

\[(\text{PERSON}) \rightarrow \text{spouse} \rightarrow \text{spouse} \rightarrow (\text{PERSON})\]
Those wonderful void references!

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?

```java
class STOP {
    ...  
    station: STATION
    invariant
        station /= Void
    ...
}
```

- Use creation procedure:

```java
create stop1.set_station(Central)
```
class STOP
create
  set_station
feature
  station: STATION
    -- Station which this stop represents
  next: SIMPLE_STOP
    -- Next stop on the same line
set_station(s: STATION)
  -- Associate this stop with s.
  require
    station_exists: s /= Void
  ensure
    station_set: station = s
link(s: SIMPLE_STOP)
  -- Make s the next stop on the line.
  ensure
    next_set: next = s
invariant
  station_exists: station /= Void
end
Object creation: summary

To create an object:

- If class has no `create` clause, use basic form:
  \[\text{create } x\]

- If the class has a `create` clause listing one or more procedures, use
  \[\text{create } x.\text{make}(\ldots)\]
  where `make` is one of the creation procedures, and \((\ldots)\) 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
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
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

Which entities are still Void after execution of line 4?
Which of the classes mentioned here have creation procedures?
Why is the creation procedure necessary?