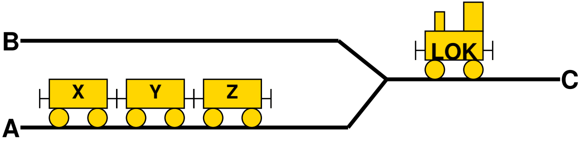
2 Problem, Algorithm, Program

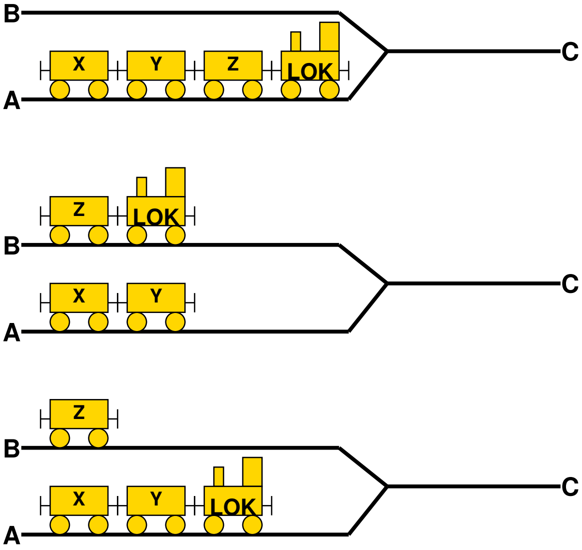
- What is a 'problem'?
- What is an 'algorithm'?
- What is a 'program'?
- Insertion: Java examples, part 1
- **Flow control of algorithms**
- Insertion: Java examples, part 2
- Structured composition of algorithms

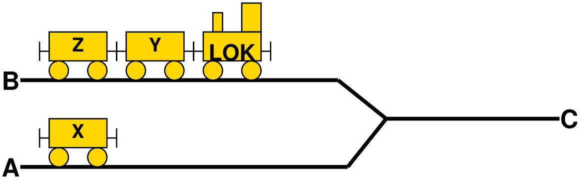
informal example: shunting algorithm

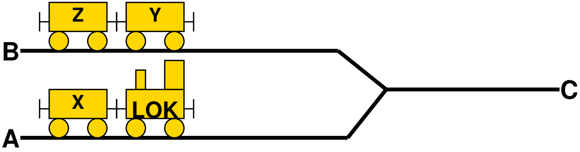
- given:
  1. railway tracks with 2 storage tracks ***A***, ***B*** and a track ***C*** with switch to ***A*** and ***B***.
  2. coaches in order ***X***, ***Y***, ***Z*** on track ***A***
  3. shunting engine ***LOK*** on track ***C***

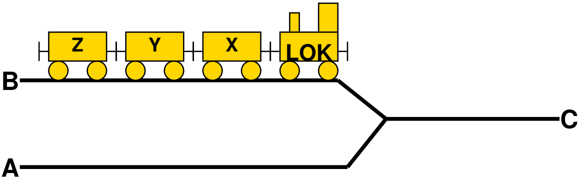


- wanted:
  - Algorithm for reordering the coaches using the engine, target order ***Z***, ***Y***, ***X*** on track ***B***.

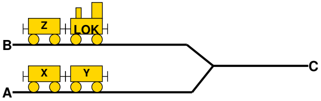

B
X
Y
Z
LOK
A
C
Z
LOK
B
X
Y
A
C
Z
B
X
Y
LOK
A
C


Z
Y
LOK
X
B
A
C

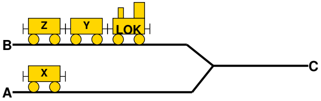

Z
Y
X
LOK
B
A
C


Z
Y
X
LOK
B
A
C

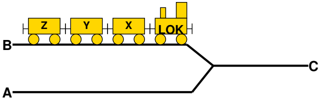
01 Move engine to track ***A*** and couple coach.
02 Uncouple next coach.
03 Move engine and coach to track ***C***.
04 Continue to track ***B***.
05 Uncouple engine.
06 Move to track ***C***.



07 Move engine to track ***A*** and couple coach.
08 Uncouple next coach.
09 Move engine and coach to track ***C***.
10 Continue to track ***B***.
11 Couple coaches and uncouple engine.
12 Move to track ***C***.



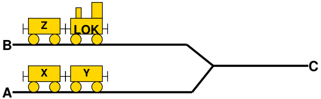
13 Move engine to track ***A*** and couple coach.
14 (no need to uncouple anything)
15 Move engine and coach to track ***C***.
16 Continue to track ***B***.
17 Couple coaches and uncouple engine.
18 Move to track ***C***.



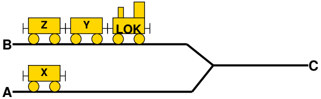
Observation 1:

$\boxed{\text{Actions are done consecutively (in sequential order)}}$

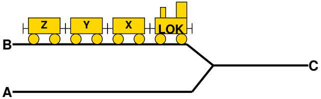
01	Move engine to track ***A*** and couple coach.
02	Uncouple next coach.
03	Move engine and coach to track ***C***.
04	Continue to track ***B***.
05	Uncouple engine.
06	Move to track ***C***.



07	Move engine to track ***A*** and couple coach.
08	Uncouple next coach.
09	Move engine and coach to track ***C***.
10	Continue to track ***B***.
11	Couple coaches and uncouple engine.
12	Move to track ***C***.



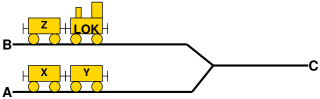
13	Move engine to track ***A*** and couple coach.
14	(no need to uncouple anything)
15	Move engine and coach to track ***C***.
16	Continue to track ***B***.
17	Couple coaches and uncouple engine.
18	Move to track ***C***.



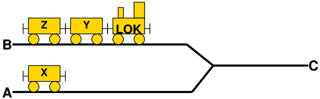
Observation 2:

Some sequences of actions are repeated

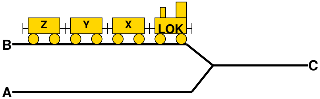
01	Move engine to track ***A*** and couple coach.
02	**Uncouple next coach.**
03	Move engine and coach to track ***C***.
04	Continue to track ***B***.
05	**Uncouple engine.**
06	Move to track ***C***.



07	Move engine to track ***A*** and couple coach.
08	**Uncouple next coach.**
09	Move engine and coach to track ***C***.
10	Continue to track ***B***.
11	**Couple coaches and uncouple engine.**
12	Move to track ***C***.



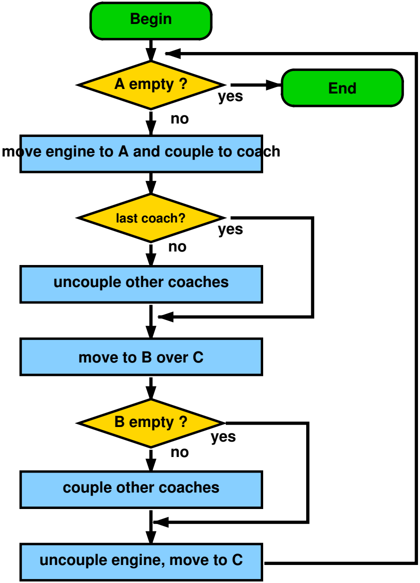
13	Move engine to track ***A*** and couple coach.
14	**(no need to uncouple anything)**
15	Move engine and coach to track ***C***.
16	Continue to track ***B***.
17	**Couple coaches and uncouple engine.**
18	Move to track ***C***.



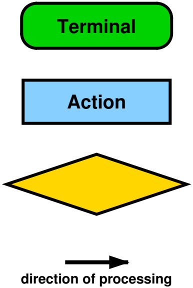
Observation 3:

Under some conditions actions are chosen from several alternatives

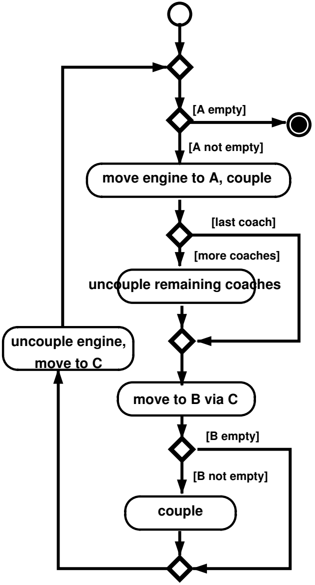
Flow diagrams to represent algorithms



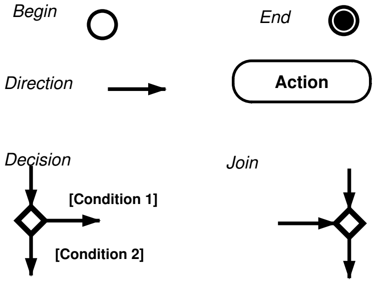
flow diagrams:
established method
to represent
Algorithms



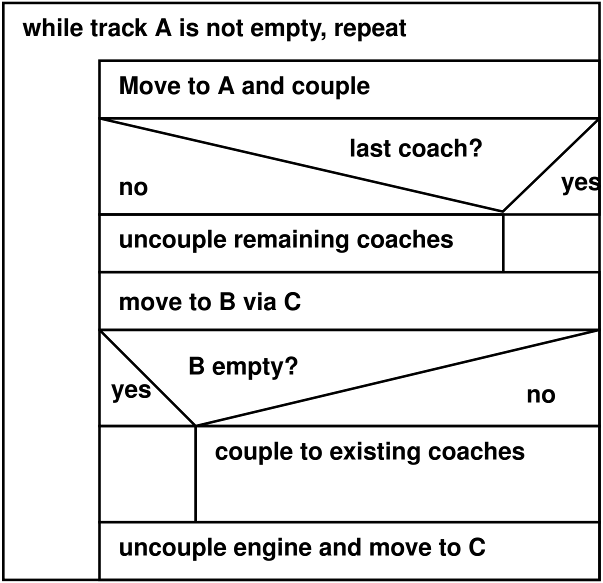
UML for representing algorithms



Unified Modeling Language (UML):
standardized graphical modelling language
for specifiying, constructing and
documenting (software) systems



Structograms for representing algorithms



2 Problem, Algorithm, Program

Shunting as a Java program

Input the desired length of the track,
then create arrays of corresponding length **`track_a`**, **`track_b`**:

```
public class K2B07E_Shunting {
  public static void main(String[] args) {
    int index;
    int count;

    int[] track_a;
    int[] track_b;

    count = Integer.parseInt(
            System.console().readLine("Count: "));

    track_a = new int[count];
    track_b = new int[count];

```

Initialize **`track_a`**,
then output:

```java
index=0;
while ( index < count )
{
  track_a[index] = index*index;
  index=index+1;
}

index=0;
while ( index < count )
{
  System.out.print(track_a[index]);
  index=index+1;
  if (index<count){
    System.out.print("-");
  } else {
    System.out.println();
  }
}
```

Assign **track_b** in reversed order,
then output again:

```
    index=0;
    while ( index < count )
    {
      track_b[index]=track_a[count-1-index];
      index=index+1;
    }

    index=0;
    while ( index < count )
    {
      System.out.print(track_b[index]);
      index=index+1;
      if (index<count){
        System.out.print("-");
      } else {
        System.out.println();
      }
    }
  }
}
```

2 Problem, Algorithm, Program

- What is a ‘problem’?
- What is an ‘algorithm’?
- What is a ‘program’?
- Insertion: Java examples, part 1
- Flow control of algorithms
- Insertion: Java examples, part 2
- Structured composition of algorithms

Controlling the flow of algorithms

The order in which the instructions of an algorithm are executed follows three basic patterns:

- **Sequence**
- **Selection** (alternative, branching instruction)
- **Iteration** (repetition, loop)

The three concepts sequence, selection and iteration are enough to formulate *all* executable algorithms.

($\leadsto$ theory of computing)

Sequence:

- At every time only one action is executed.
- Every instruction is executed exactly once.
- The order of the execution corresponds to the given sequence of instructions.
- The sequence ends once all instructions were executed.
- Every sequence consists of at least one instruction.

The sequence is the basic module of every algorithm, i.e., every algorithm is a sequence of instructions.

Selection (alternative, branching instruction):

- general form (if-else-condition):

if **condition** *then* **sequence1** *else* **sequence2**

  If the logical expression **condition** evaluates to '**true**' , then **sequence1** is executed, otherwise **sequence2.**
- special case (if-condition):

if **Condition** *then* **sequence**

  Corresponds to an if-else-condition with a sequence2 which consists only of an 'empty' instruction.

Every instruction within a sequence can be a selection, i.e., selections can be 'nested'.

Iteration (repetition, loop):

- while-loop:

$$\boxed{\textit{while}\ \textbf{condition}\ \textit{execute}\ \textbf{sequence}}$$

(1) First the logical expression **condition** is evaluated.
(2a) If the evaluation yields the result '**true**',
sequence is executed.
Then the execution continues at (1).
(2b) If the evaluation yields the result '**false**',
the repetition stops.

Every instruction within a sequence can be an iteration, i.e., iterations can be 'nested'.

- Sequence, selection, and iteration define **sequential algorithms**.
- If the flow control is extended by means to split (**fork**) and recombine (**join**) sequences executed in parallel, then **parallel algorithms** are possible.
- Sometimes parallel algorithms allow for a faster solution of problems, but they are not more powerful than sequential algorithms, i.e., they can solve the same problems.

Typical properties of algorithms

- **Finiteness**: The description of an algorithm has a finite length. During processing an algorithm, the created data structures and intermediate results are finite.
- **Termination**: The algorithm produces a result after a finite number of steps.
- **Abstraction**: An algorithm solves a class of problems; the actual problem instance to solve is determined by the input data.
- **Determination**: Algorithms are usually determined, i.e., they always return the same output values for the same input values each time they is executed.
- **Determinism**: An algorithm is deterministic, if there is at most one possible continuation at each step of its execution.

Structured Algorithms

An algorithm is **structured** if it is constructed following this 'grammar':

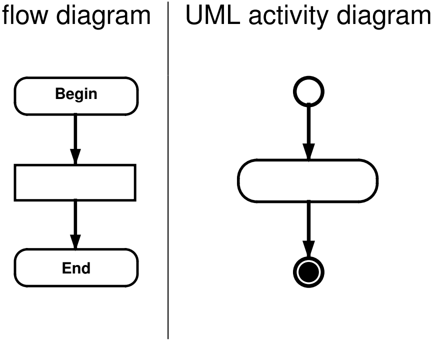
algorithm	$\longleftarrow$	sequence
sequence	$\longleftarrow$	instruction
		| instruction sequence
instruction	$\longleftarrow$	simple_instruction
		| selection_from_sequences
		| iteration_of_a_sequence
simple_instruction	$\longleftarrow$	...

(Example how to read this: a sequence consists of ('$\longleftarrow$') an instruction or ('|') an instruction, followed by a sequence).

In a structured algorithm every instruction has exactly one entrance and one exit.
(goal: avoid intransparent processes, no 'spaghetti code')

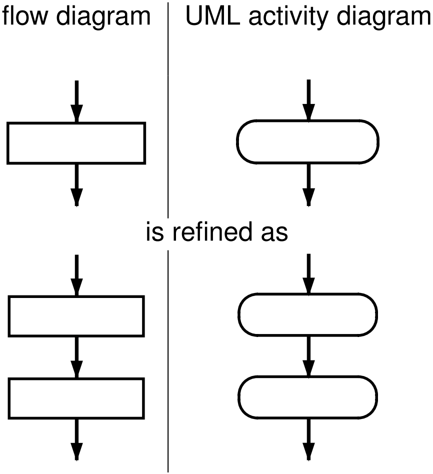
Construction of structured algorithms by ‘refinement’

For a top-down construction of an algorithm we start with the most simple diagrams:

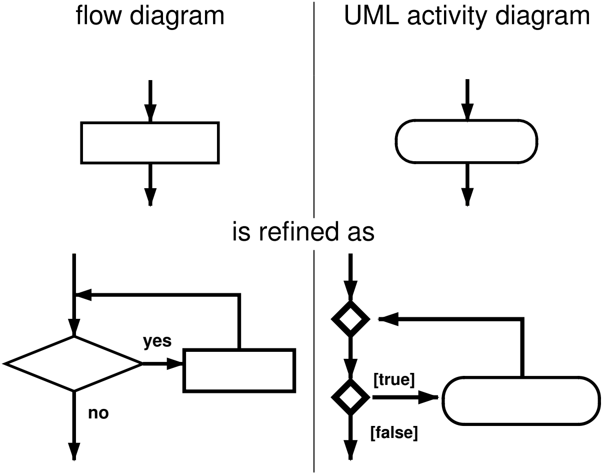


Goal: every structure has exactly one entrance and exactly one exit...

A sequence can consist of a sequence of instructions:

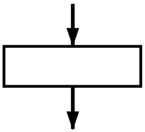


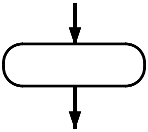
An instruction can consist of an iteration:



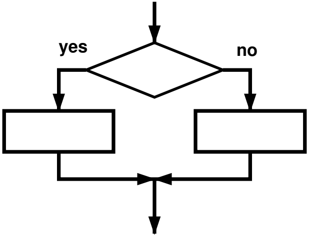
An instruction can consist of a selection:

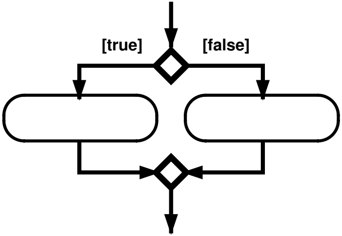
flow diagram | UML activity diagram





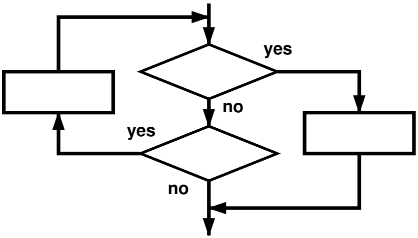
is refined as





Not every flow diagram or UML activity diagram is structured!

For example the following diagram cannot be constructed as shown before:



- The restriction to structured algorithms restricts the options to construct algorithms in favour of a better readability!
- For every flow diagram there is a structured flow diagram that computes the same result (see lectures on theory of computing).