DRAKON

The Human Revolution in Understanding Programs

October 2011
Stepan Mitkin
stipan.mitkin@gmail.com

A Graphic Language

DRAKON (Дракон) is a graphic language for explaining algorithms.

It is a tool for easy and *fast* understanding between human beings when they talk about programs. DRAKON's slogan is “took a glance – got the idea”.

It Comes from Space

DRAKON was invented within the Russian space program for programming the AI that controlled the Buran spacecraft. The Buran developers soon realized that the **main** problem in building a large system was the **human** factor and **communication** between humans. Explaining an algorithm turned out to be much harder than designing and implementing it. The *Keldysh Institute for Applied Mathematics* in Moscow was given the task to address that problem. Their response was the DRAKON language.

Understanding is Key To Software Process

Clear, easy to read programs and documentation are the bedrock of a good software process.

- When a program is easy to understand, **errors** inside it become **visible** before the program runs. The value of seeing errors early cannot be overestimated. First, there is no need to spend long hours on debugging and testing. Second, when an error indicates a flaw in the design, that flaw has more chances to get fixed when discovered shortly after the project start. As a result, better software is delivered within a shorter time period.
- A large software project is nothing but a cloud of information being constantly grown by many people. Each individual contribution must fit the existing environment, that is why there is more reading from that cloud than adding to it. New requirements must go along well with the old requirements, new code must work nicely with the code that has already been there. **Understanding** what **others** have done is **essential** for work in a large project. That is why people co-operate more efficiently if their project is easy to understand.

Clarity means *success*, while obscurity means *failure*.

Understanding Must be Made Easier

What is the **problem** with understanding?

1. Programs are getting more **complex**.
2. They are getting **harder** to understand.
3. Understanding is **work**.
4. Productivity of understanding is intolerably **low**.

Conclusion: understanding must be **made easier**.

How to Improve Understanding?

DRAKON's revolutionary way of making algorithms easy to understand is the **combination** of mathematical strictness **and** human ergonomics.

One one hand, DRAKON is mathematically strict which means that:

1. It is precise and unambiguous.
2. It has been proven that DRAKON can express **ANY** algorithm.

On the other hand, mathematics alone is not enough and that is why DRAKON is so centered around usability and ergonomics.

What is ergonomics? Ergonomics is the art of making **exhausting** and boring work *easy* and comfortable. It is about making things convenient to use by human beings. Here is the DRAKON approach to ergonomics:

1. DRAKON is **simple**.
2. DRAKON is a **graphic** language.
3. DRAKON's graphics notation is highly **optimized**.

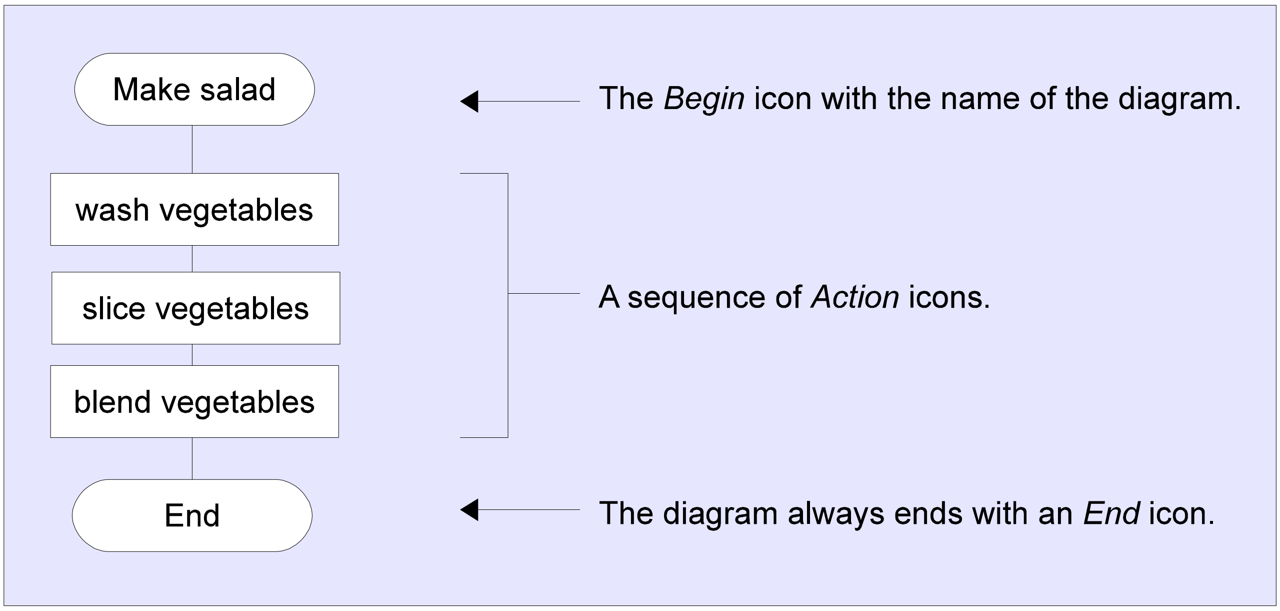
Why Graphics?

Text is the primary means of capturing algorithms at this time. Both requirements and code are forms of text, and everybody seems to take the **domination** of text as a natural and **unavoidable** fact. Diagrams are used occasionally and inconsistently. This situation is a real problem:

1. The human eye and brain are very well **optimized for visual** information which constitutes most of information received by man. We are bad at reading and good at seeing pictures. In order to be productive, we must use our body and mind in the way they are supposed to function.
2. An image is a more **compact** representation of information than a text. This is especially true when a high level of detail is required.
3. **Engineers** switched from text to graphics long, long ago with a great success. They have built the whole modern technological world using drawings and blueprints. Why are software developers considered different?

The future belongs to *graphic* languages because they are *better suited* to human beings.

Fig 1. The simplest DRAKON diagram.


Make salad
wash vegetables
slice vegetables
blend vegetables
End
The *Begin* icon with the name of the diagram.
A sequence of *Action* icons.
The diagram always ends with an *End* icon.

How Does DRAKON Optimize Graphics?

Why need yet another graphic notation? What is so unique about DRAKON?

1. Elements on a DRAKON chart are arranged in a **hierarchical** way. You see the most important things first.
2. DRAKON takes into account human **visual habits** and minimizes the effort required to scan a diagram. Your eyes follow a natural and repeating pattern without the need to jump around and look for things.
3. DRAKON's **strict rules** and guidelines make sure that the diagram does not turn into clutter. Order is always guaranteed, the lines and arrows never become a cobweb.

Most Important Things First

The DRAKON diagram has several vertical sections called *branches*. The branches represent a logical decomposition of the problem.

The first glance at the DRAKON diagram tells the summary of the diagram and answers the Three Questions of the King:

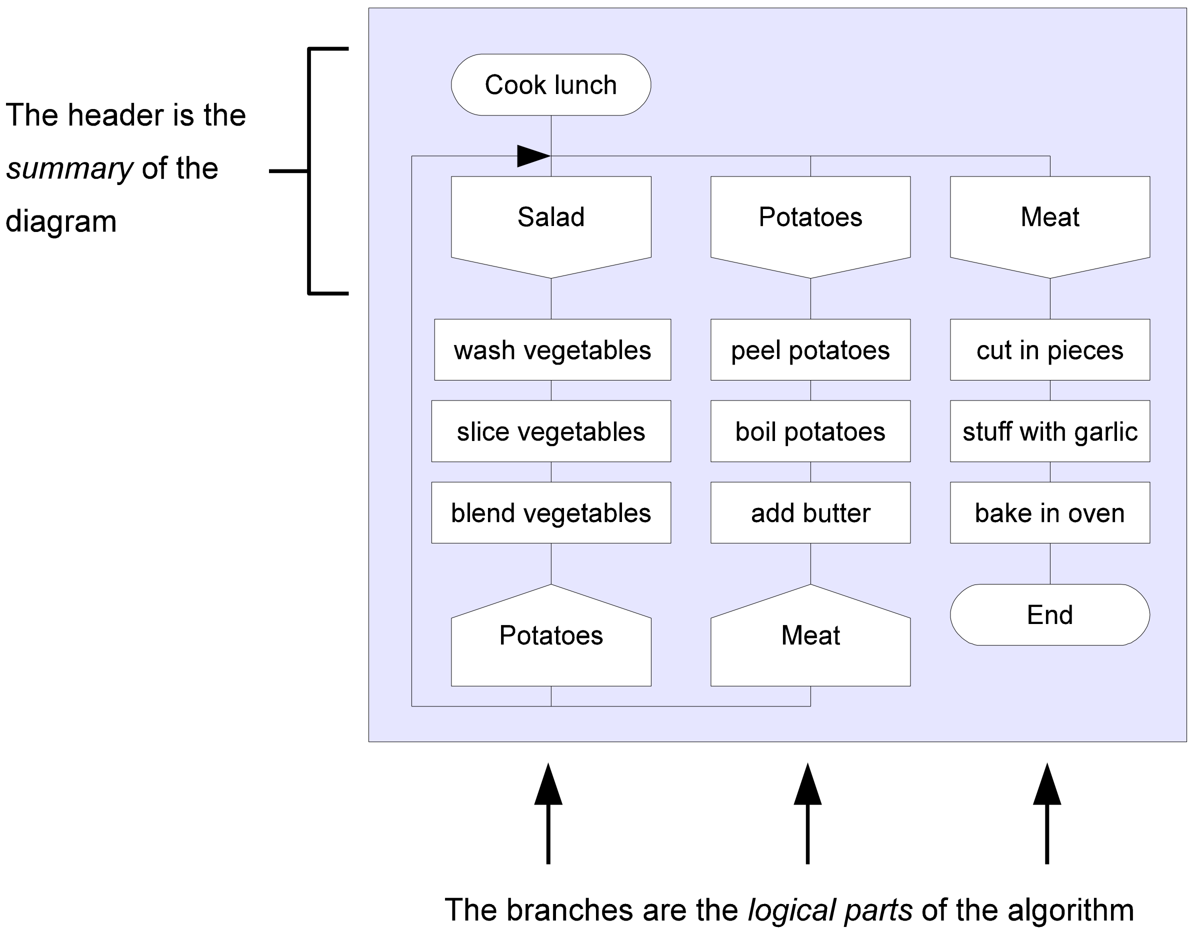
1. What is the **name** of the problem?
2. How many **parts** does the problem have?
3. What are the **names** of the **parts**?

The *Begin* label contains the name of the diagram and answers the first question.

The number of branches answers the second questions. The names of the branches answer the third question and tell about the main logical parts of the problem.

The branch names together with the *Begin* label are placed in the *header* of the diagram, which ensures that the summary of the diagram can always be found at the same place.

Fig 2. Branches and header in the DRAKON diagram



Branches

A *branch* represents a sequence of actions. The actions are recorded in the branch in the top-down direction. This direction is very important in our world because gravity forces objects to move down. This is why downward movement is accepted as "natural" by our eyes.

Rule: A branch has one entry and one or more exits.

The entry is a special icon at the top that holds the branch name. Branch names must be unique within a diagram. The branch entry is the only point where execution of the branch can start.

Branch exits are located at the bottom of the branch. There are two kinds of branch exits:

- An *Address* icon that shows the name of the next branch.
- The *End* icon that marks the end of the algorithm.

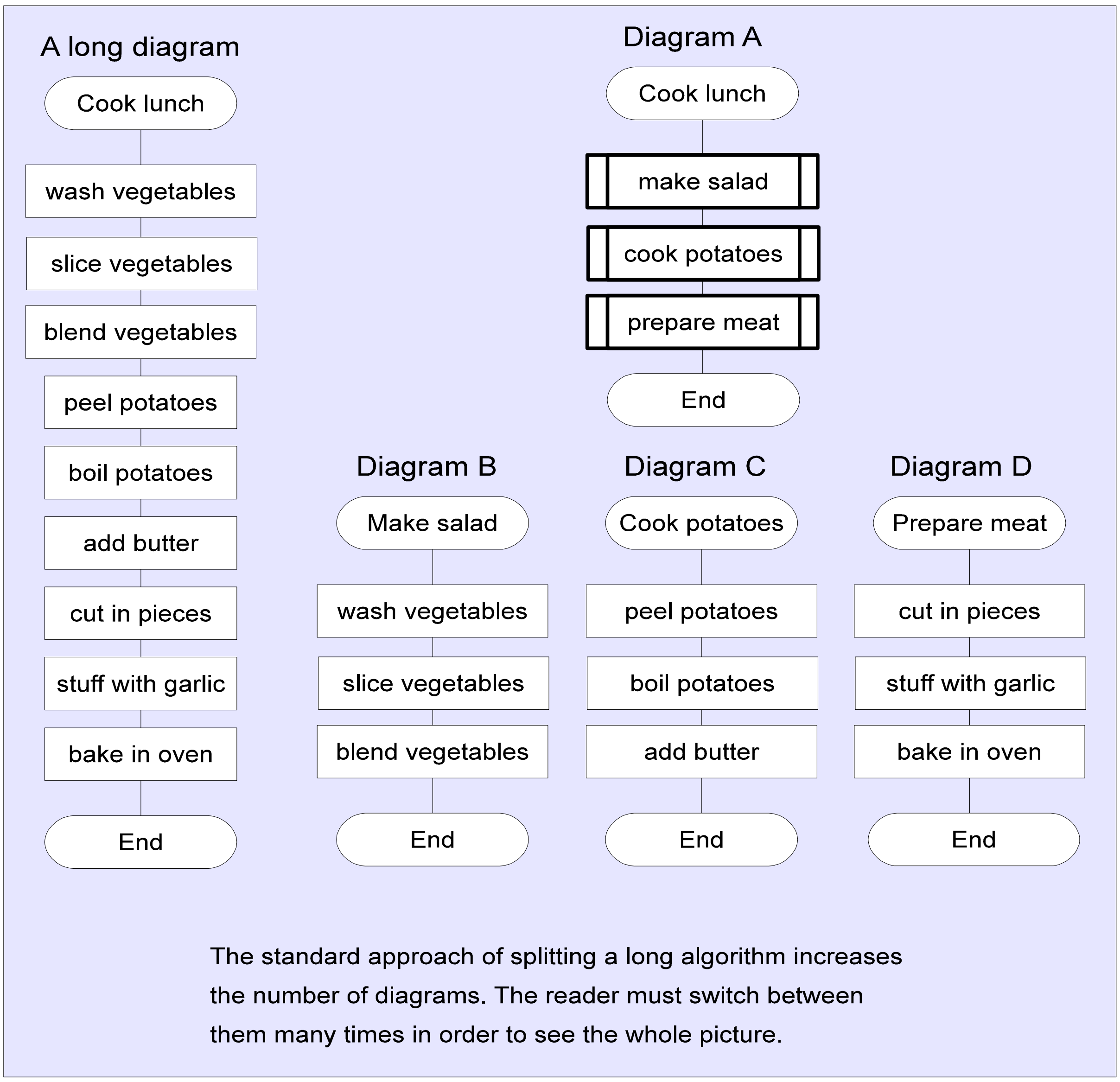
Why branches?

Long algorithms are hard to understand because the human mind is not good at dealing with too many objects at the same time. Dividing an algorithm into several conceptual parts helps improve readability.

With the divide and conquer approach, a long and complex structure can be viewed at any level of detail. Usually this is done by recursively splitting a flat list of actions into sub-routines.

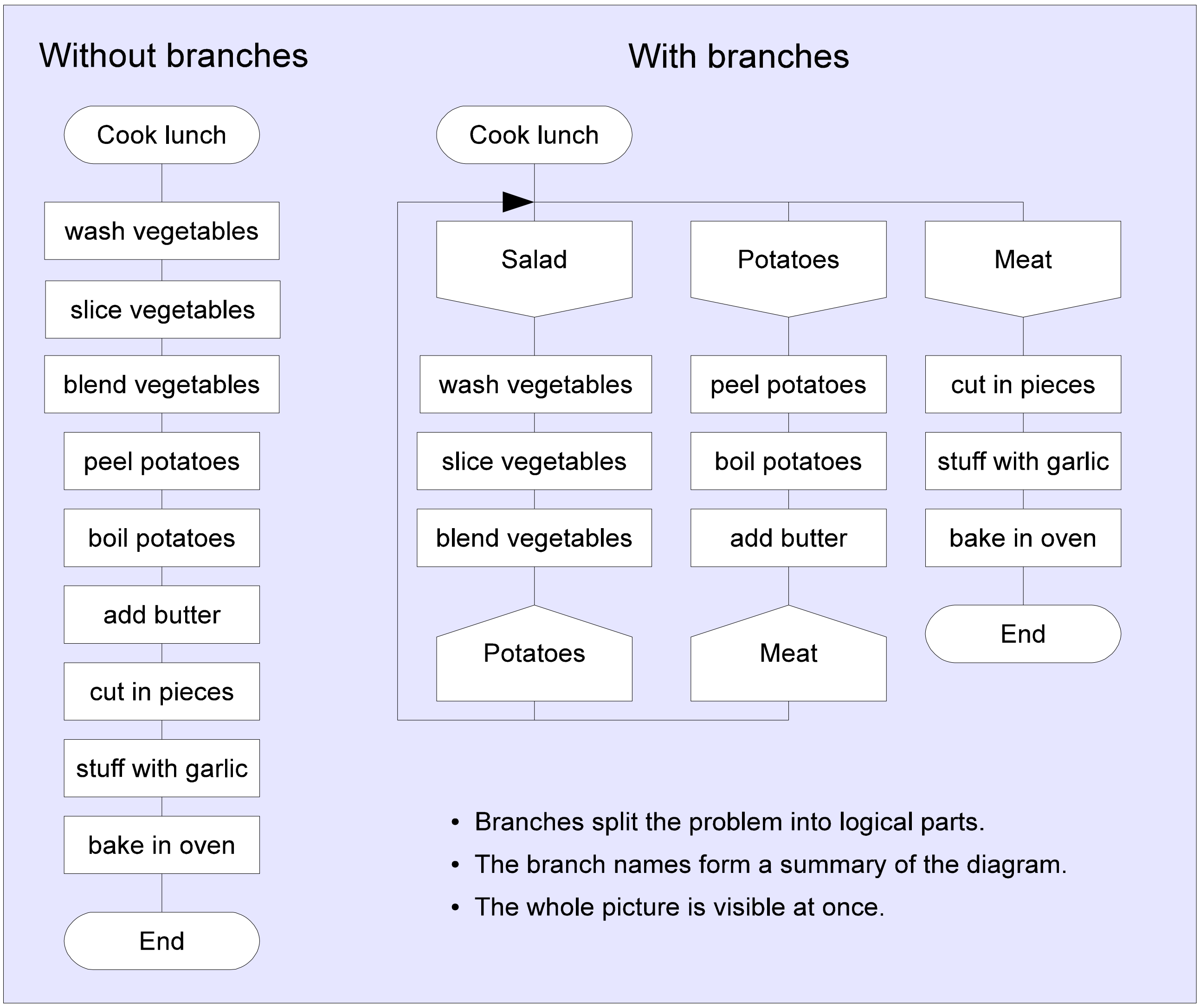
DRAKON adds a unique addition to that technique: with DRAKON branches, you can see **two levels** of this subdivision on the **same** diagram, while a sub-routine only shows items on one level.

Fig 3. Splitting a long algorithm into sub-routines.



The standard approach of splitting a long algorithm increases the number of diagrams. The reader must switch between them many times in order to see the whole picture.

Fig 4. Using branches to split a long algorithm.



How Branches Work

Functioning of a DRAKON diagram can be visualized with a runner moving along the wires:

1. The runner goes down through the leftmost branch.
2. Then it goes to the left edge and climbs up to the left top corner.
3. Then it slides to the right until it finds the branch pointed to by the *Address* icon of the previous branch.
4. It goes down through that branch.
5. It finds the next branch like before.

Jumps between branches, when the runner moves from one branch directly to the middle of another, are not allowed.

Order of Branches

Branches on the DRAKON diagram are ordered from left to right according to their sequence in time.

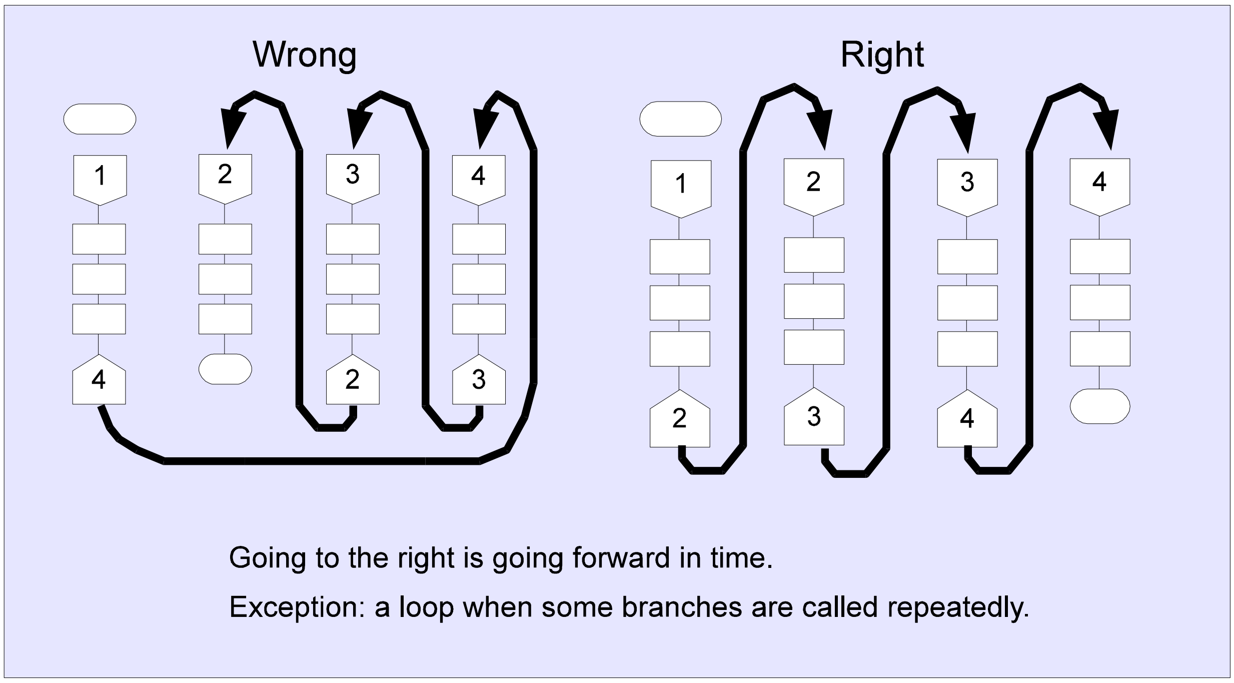
Rule: going to the right is going forward in time.

Following this rule ensures that the branch position in space reflects its position in time. The *Address* icon at the bottom of a branch points to the next branch and that potentially allows for any ordering of branches. But DRAKON considers legal only two cases when an *Address* icon does not point at the next branch to the right:

1. When one or more branches must be skipped. This can only happen as a result of evaluation of a conditional statement (like *if* or *switch*), we can't jump over branches just for fun.
2. When some branch should be executed more than once. This is a special case of loop, called the "branch loop".

A diagram that follows the above guidelines is ergonomic because it is arranged in a consistent and natural way, like columns of text in a newspaper.

Fig 5. The order of branches on a diagram.


Wrong
Right
1
2
3
4
4
2
3
1
2
3
4
2
3
4
Going to the right is going forward in time.
Exception: a loop when some branches are called repeatedly.

The Skewer

A *skewer* is a sharp metal stick that is used for roasting meat. Skewer illustrates the main principle of arranging elements inside a branch:

1. The entry and the main exit of a branch are connected by a straight vertical line.
2. The icons that comprise the main path of a branch lie on that vertical line.

When this rule is met, the diagram becomes cleaner looking and better organized. In the absence of a skewer, the branch looks broken and ugly.

Fig 6. Skewers.

The Main Route of a Branch

If a branch contains conditional icons like *if* and *switch*, there is more than one path through it. In this case we could call one path as *the main route* and others as *secondary routes*.

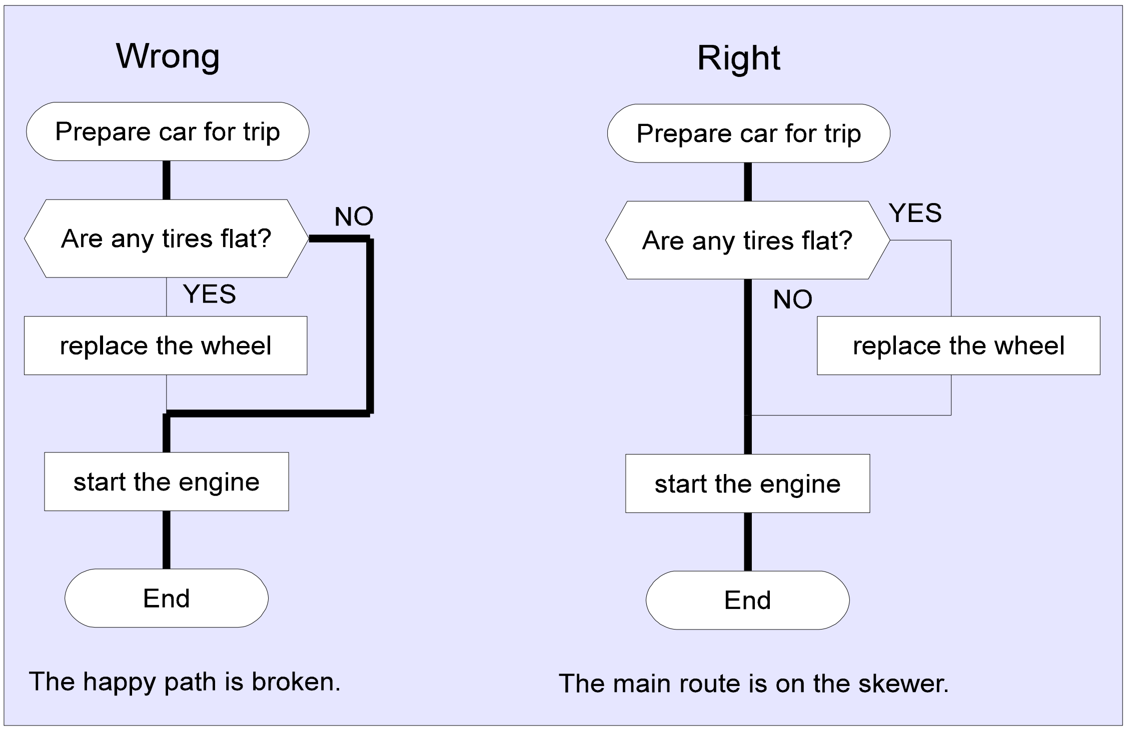
The man route is the path that leads to the greatest success. In other words, it is the **happy path** of the algorithm. It is the way everything is supposed to work.

It should be possible to find the main route of a branch with just a single glance. The main route must be immediately noticeable.

Rule: The main route must lie on the skewer.

The exits of the *if* and *switch* icons in the branch should be swapped in such a way that *the main route* follows the leftmost vertical of the branch.

Fig 7. The main route must be straight and go on the skewer.



Order of Secondary Routes

A *secondary route* is any path in the algorithm besides *the main route*. Secondary routes are placed to the right from *the main route*.

The rule of secondary routes: the further to the right – the worse it is.

This means that the further far away a route is from the main route, the less pleasant situation it describes.

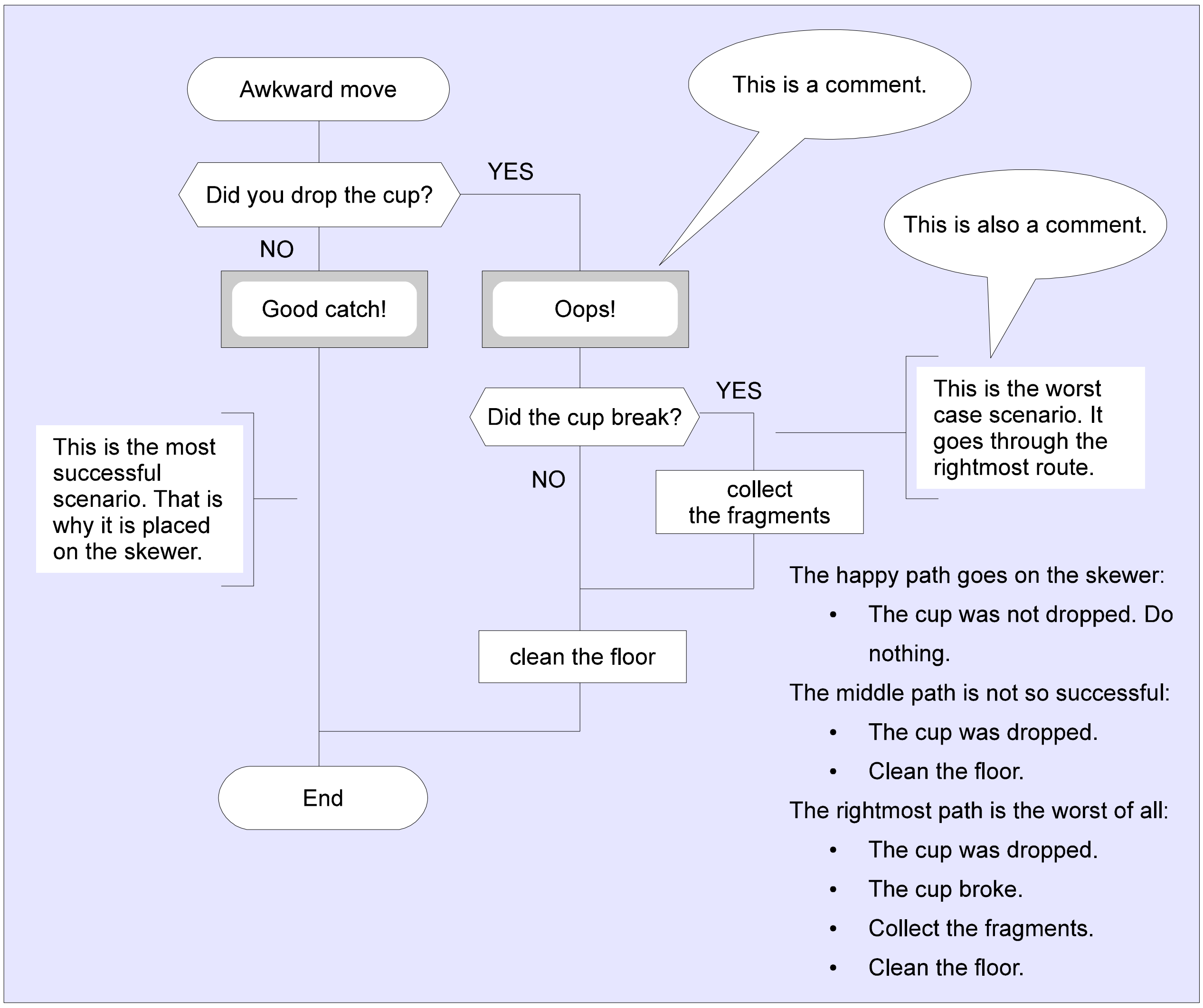
One of the reasons why DRAKON was created is that the traditional flowcharts are totally horrible as far as ergonomics is concerned. They are extremely hard to figure out. On the other hand, DRAKON diagrams ensure order and clarity by providing a consistent visual structure. This structure is additional, “redundant” information about the algorithm, a **presentation layer** that makes it easy to understand.

What if all the paths are equally successful? Then we should come up with some other ordering to sort the routes. For example:

- the further to the right – the more far away.
- the further to the right – the higher.
- the further to the right – the faster.
- the further to the right – the heavier.
- the further to the right – the more expensive.

The main idea here is to select a criterion which suites the situation and consistently apply it to arrange the routes.

Fig 8. The rule of secondary routes in the tea spill example.

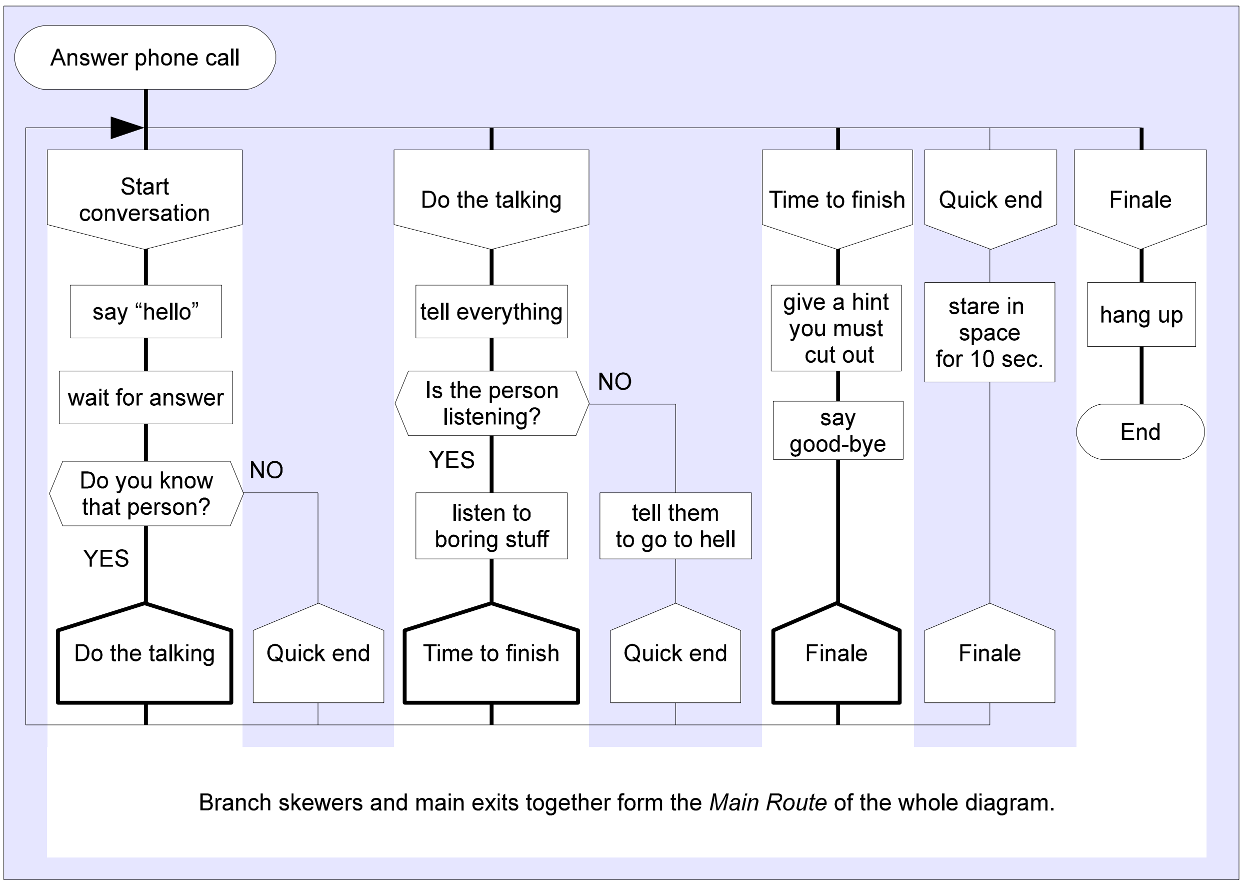

Awkward move
This is a comment.
Did you drop the cup?
YES
NO
This is also a comment.
Good catch!
Oops!
Did the cup break?
YES
NO
This is the worst case scenario. It goes through the rightmost route.
This is the most successful scenario. That is why it is placed on the skewer.
collect the fragments
clean the floor
End
The happy path goes on the skewer:
• The cup was not dropped. Do nothing.
The middle path is not so successful:
• The cup was dropped.
• Clean the floor.
The rightmost path is the worst of all:
• The cup was dropped.
• The cup broke.
• Collect the fragments.
• Clean the floor.

The Main Route of The Diagram

A secondary route may either come back to its parent route or end the branch with its own *Address* icon. Since there can be several secondary routes, the branch may have several secondary exits. *The main exit* is the exit at the lower end of the skewer. The rule of secondary routes applies to the exits of the branch as well.

The rule of secondary exits: the further an exit is to the right from the main exit, the worse. This rule makes it possible to grasp the general idea of the algorithm with a single glance. The main exits from the branches connect the main routes of the branches and form *the main route of the diagram*. This way we can easily see the happy path of the whole algorithm and leave the details for later.

Fig 9. The Main Route of the diagram.



Many Entries, One End

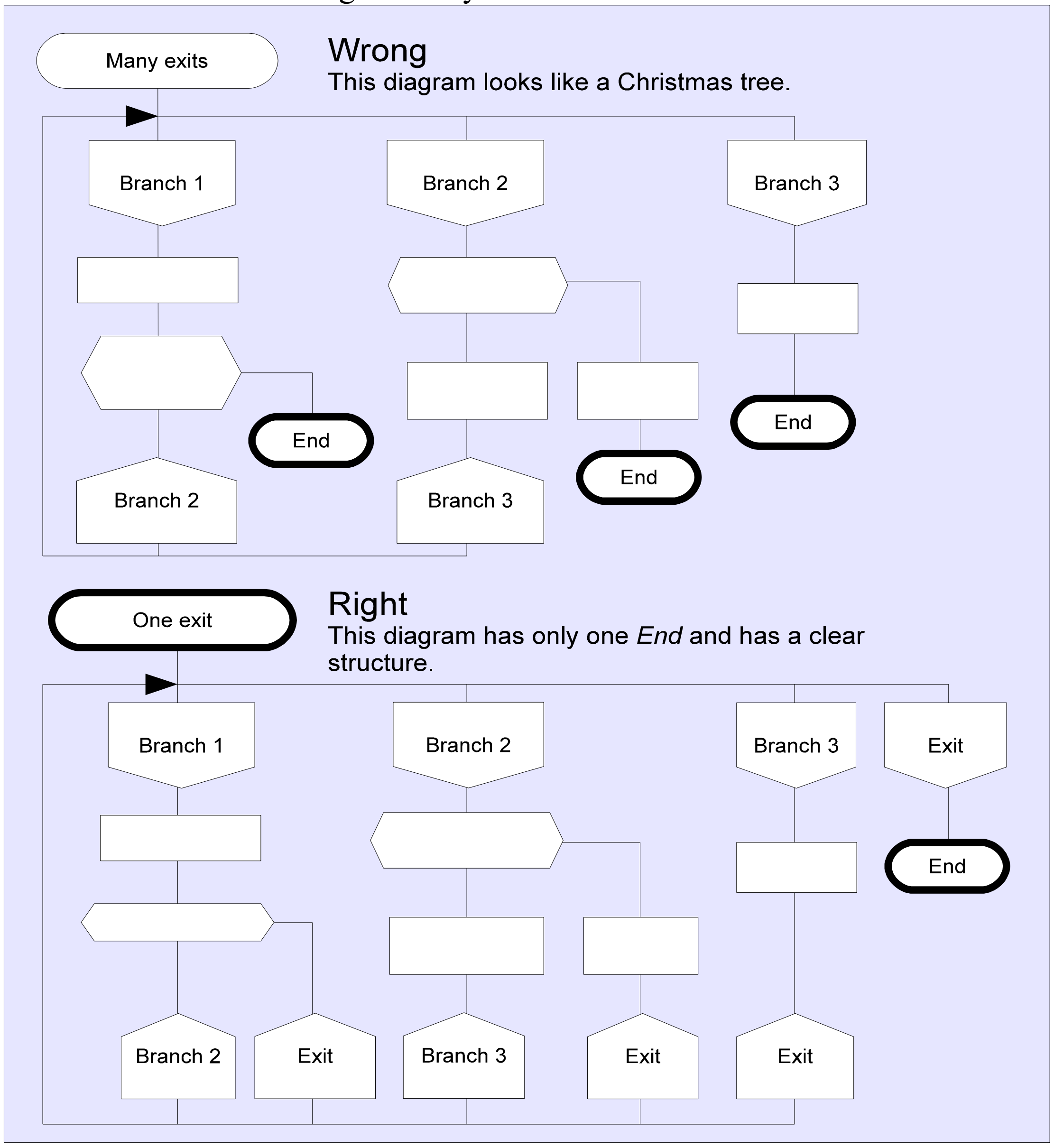
A diagram can have more than one entry. Sometimes, we need to skip a few branches at the beginning and proceed directly to the right part of the diagram. It is possible to do so by placing an additional *Begin* icon over the branch that we want to start with (see Fig. 26).
A diagram, however, cannot have many *End* icons. There still can be early exits from the algorithm, but they should be implemented as jumps to the last branch.

Rule: there can be only one exit.

Why have such a limitation?

- With exactly one end, the diagram has a clear structure: it goes from the "before" state at the top-left to the "after" state at the bottom-right. The possibility to have many entries contradicts this principle somewhat, but at least the entries can be easily found – they are at the top.
- It becomes easy to trace the sequence of branches from both the beginning and the end if here is only one end.

Fig 10. Only one exit is allowed.



The If Icon

The *If* icon has one entry, but two exits. The exits are marked with labels *yes* and *no*.

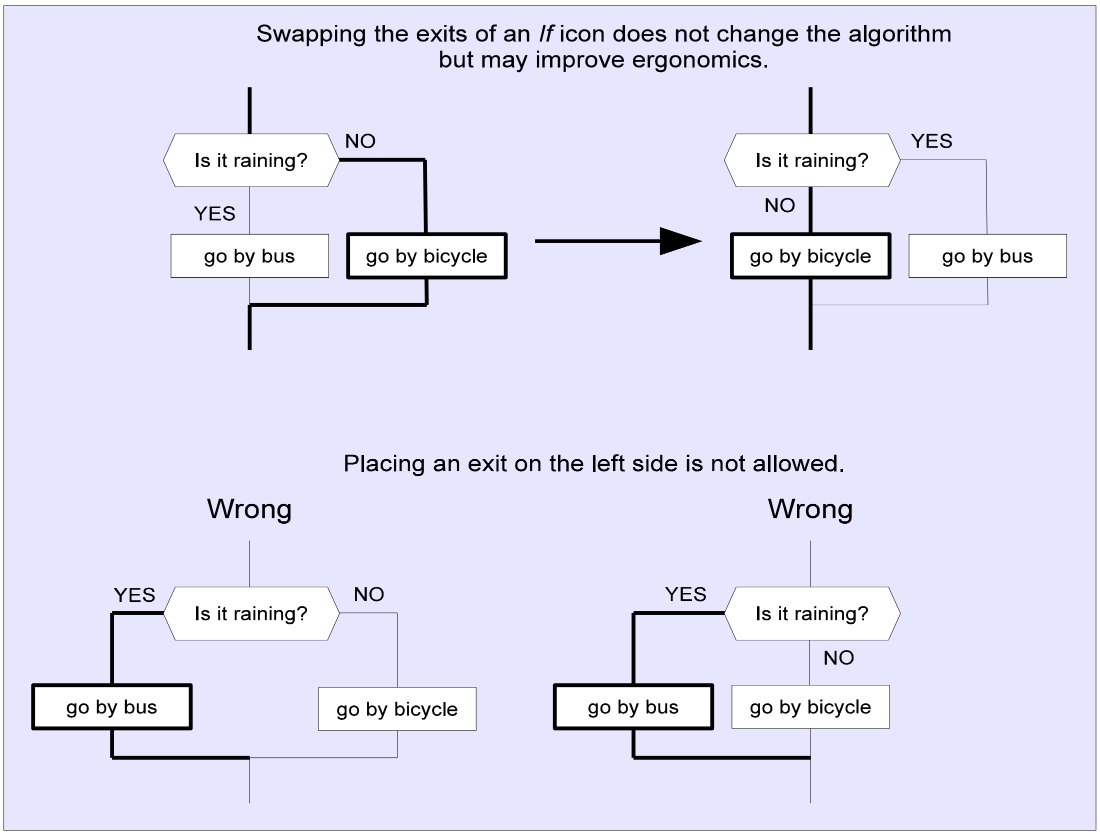
- The central exit comes out of the bottom of the icon, the right exit comes out of its right side.
- Placing an exit on the left side is not allowed.
- It does not matter which exit has the label *yes*, it could be either central or right.
- Use *yes* instead of "true", *no* instead of "false". Children learn *yes* and *no* at a very early age, *yes* and *no* are intuitive.

Note that the *If* icon is a hexagon, not a diamond like its **flowchart** counterpart. The hexagon shape saves vertical space on the diagram.

Improving Ergonomics

Swapping the central and the right paths that come out of the *If* icon is an important tuning technique. It does not change the algorithm of the diagram, but may improve its ergonomics by following the path ordering rule (the further to the right, the worse it is) and the main route rule (the main route must lie on the skewer).

Fig 11. Using the If icon.



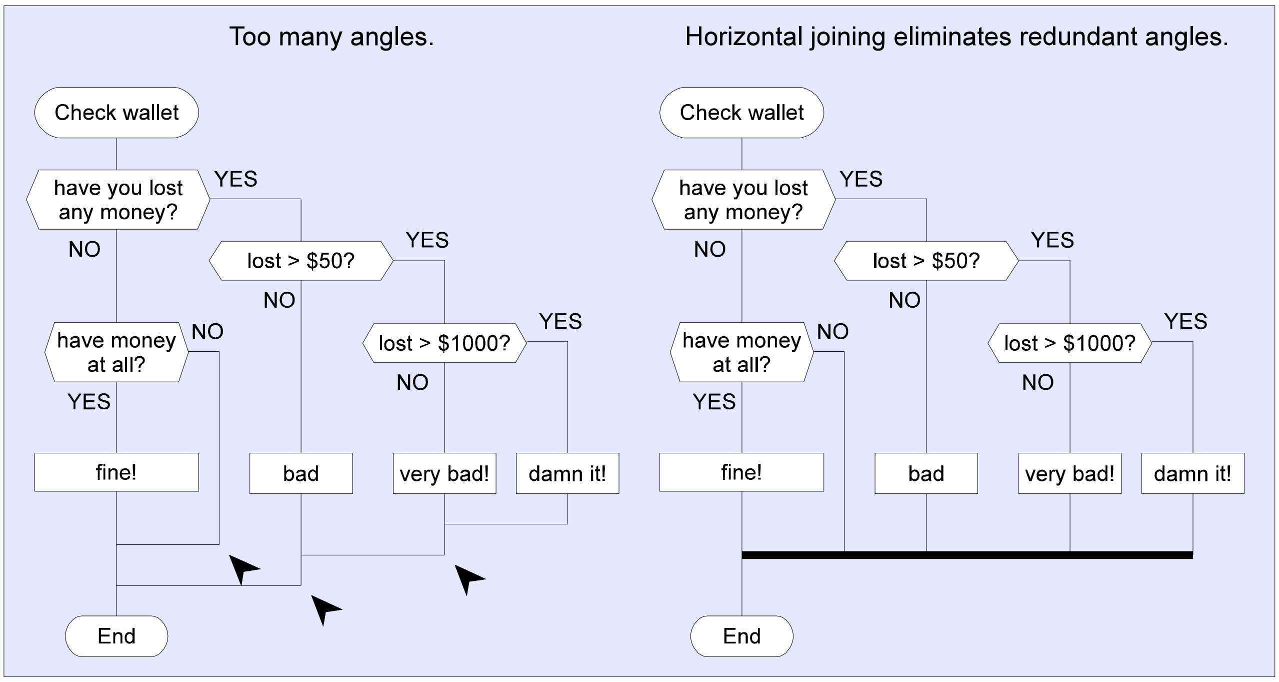
Joinings

Another way to improve ergonomics is to eliminate duplication of icons and groups of icons.

Rule: do not repeat yourself.

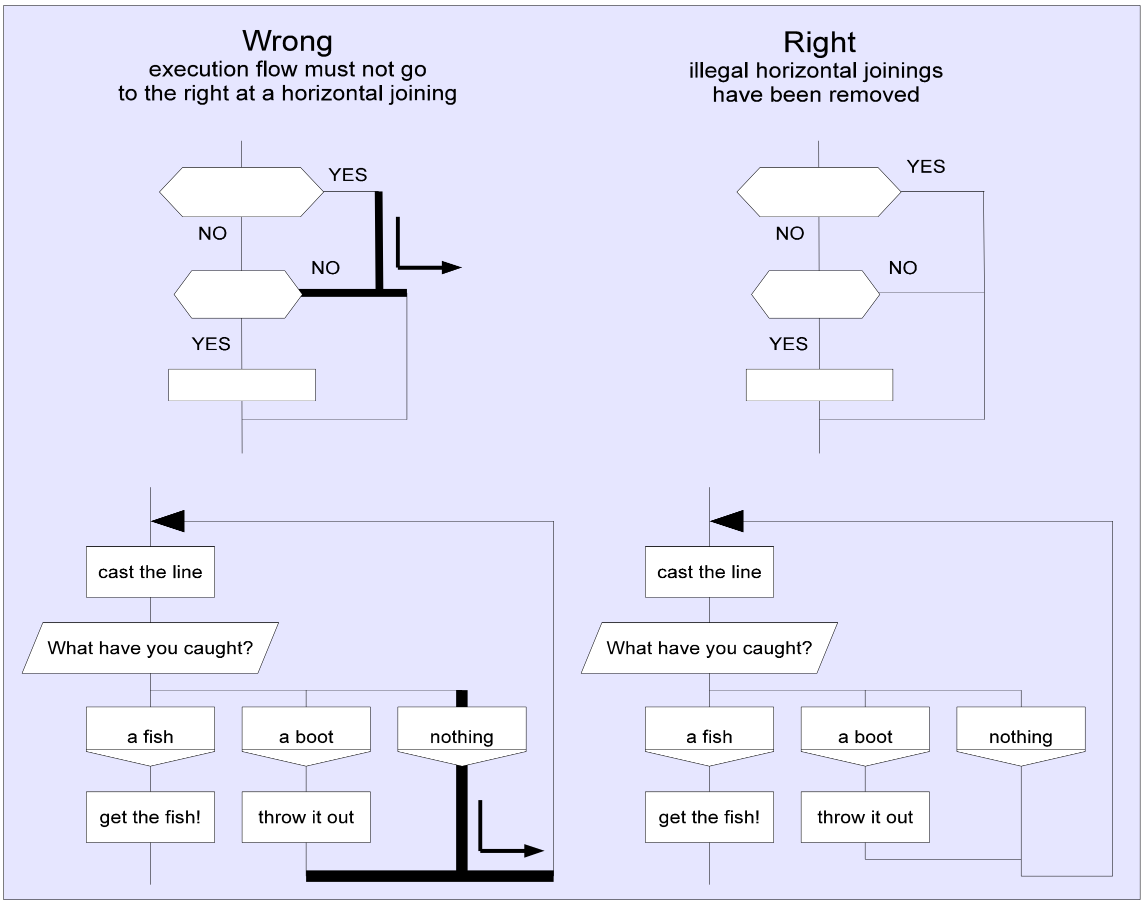
Copy-paste on the diagram is as bad as it is in the code. There are techniques to avoid duplication: *vertical* and *horizontal joining*.

Fig 12. Horizontal joining.



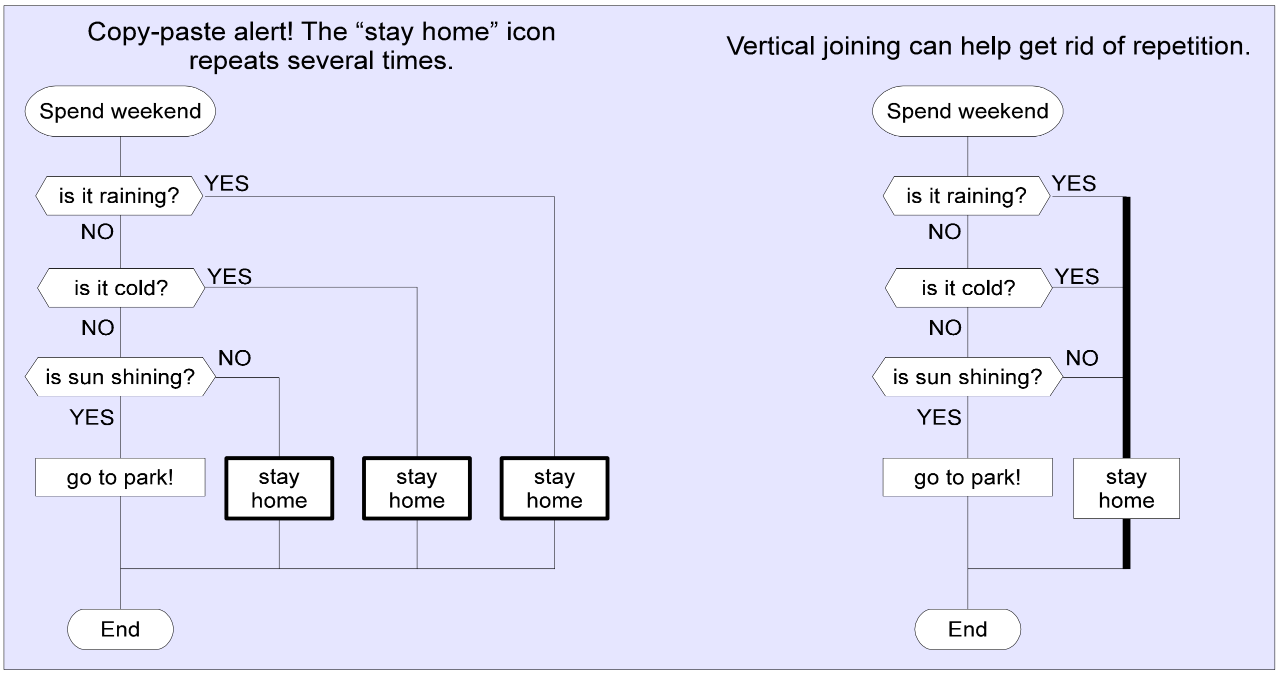
At a horizontal joining, the vertical line goes down until it hits the horizontal line. The execution flow in the horizontal line can go in either of two directions: to the left or to the right. A horizontal joining is only allowed if the execution flow goes to the left on the horizontal line. The rationale behind this is simple: the reader does not need to think which way to go after hitting a horizontal line from above. It is *always* to the left.

Fig 13. Illegal horizontal joinings.



Rule: after a horizontal joining the execution flow goes to the left.

Fig 14. Vertical joining.

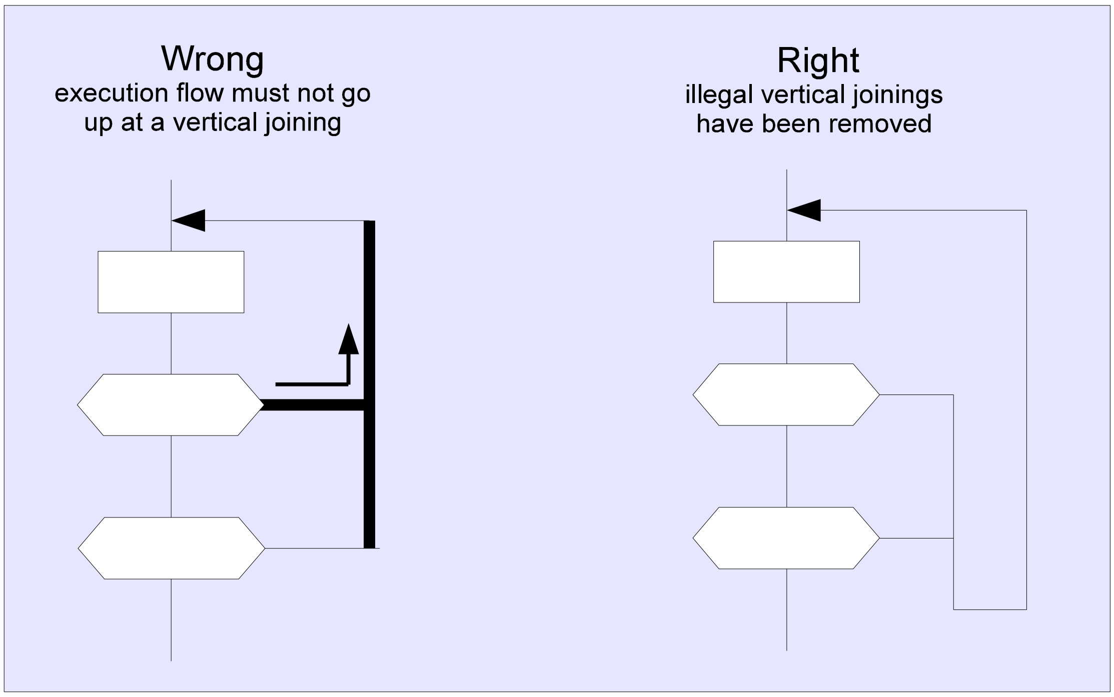


A similar one-direction rule applies to vertical joinings. Hitting a line that goes up is not allowed.

Rule: after a vertical joining the execution flow goes down.

The explanation is also similar: the reader's eyes should not jump all over the whole diagram in order to figure out whether the line goes up or down. It is *always* down.

Fig. 15. Illegal vertical joining.


Wrong
execution flow must not go up at a vertical joining
Right
illegal vertical joinings have been removed

No Line Intersections!!!

Traditional flowcharts have long been hated for their tendency to quickly turn into an entangled cobweb of lines and arrows. A dense grid of connecting lines makes reading a diagram harder than reading source code. This is why diagrams in general are often neglected by programmers who prefer pseudocode for explaining algorithms.

Line intersections are the main source of clutter in diagrams.

Rule: line intersections and breaks are not allowed.

All types of line intersections are considered ergonomically harmful and therefore illegal. In order to ensure clarity, DRAKON avoids unnecessary detail and visual noise.

The ban on intersections is a serious topological limitation. Does it prevent expressing complex real-life algorithms? No. It has been mathematically proven that DRAKON can express *any* possible algorithm *without* line intersections.

If there is a **contradiction** of rules on a particular diagram, we prioritize the rules in the following order:

1. Always follow the basic rules of DRAKON: no line intersection, no exits on the left side of the *If* icon, etc.
2. Follow the main route rule.
3. Minimize the number of angles on connecting lines.
4. Minimize the number of vertical lines.

Although it might be tempting to reduce the number of elements on the diagram at the cost of violating the ergonomics rules, it should never happen.

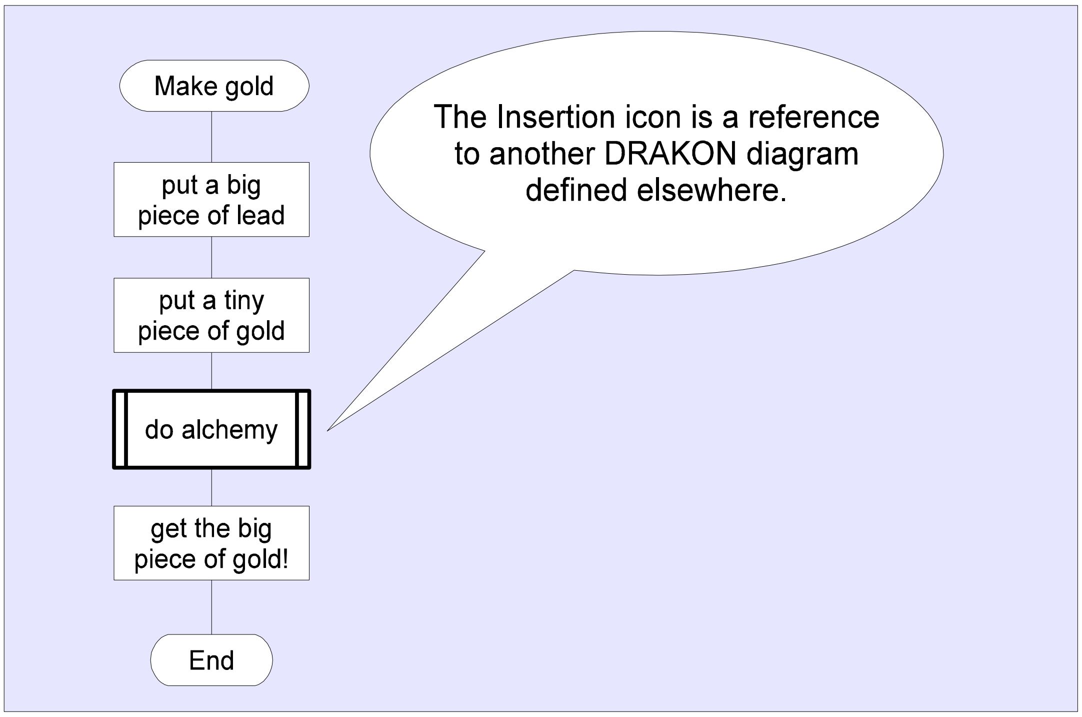
The Insertion Icon

Sub-routines in software perform two functions:

1. **Code re-use** within a single program. A sub-routine is a piece of code with a name which can be called from several places of the program.
2. **Logical** algorithm **decomposition**. Even when some code is called from only one place, it still might make sense to take it out into a sub-routine. The name of the sub-routine will explain what that code does.

The *Insertion* icon is the DRAKON notation for calling another DRAKON diagram as a sub-routine.

Fig 16. The Insertion icon.


Make gold
put a big piece of lead
put a tiny piece of gold
do alchemy
get the big piece of gold!
End
The Insertion icon is a reference to another DRAKON diagram defined elsewhere.

The For Loop

The *For* cycle is the cleanest form of loop. The *For* icon is similar to the *for* and *foreach* keywords in other programming languages. The *For* icon is actually two icons: *Begin For* and *End For*. The usual scenarios for using the *For* icon are:

1. iterating over a collection: `foreach (var item in megaList)`
2. counter-based loops: `for (int i = 0; i < length; i++)`.

The code that runs several times is represented by the icons placed between the *Begin For* and *End For* icons. Here are the rules of entering and exiting a *For* loop:

1. The *Begin For* icon is the **only single entry** into the loop body.
2. There can be **several** additional, early **exits** from the loop body.

Early exits come from the conditional icons: *If* and *Switch*.

Fig 17. The For loop.

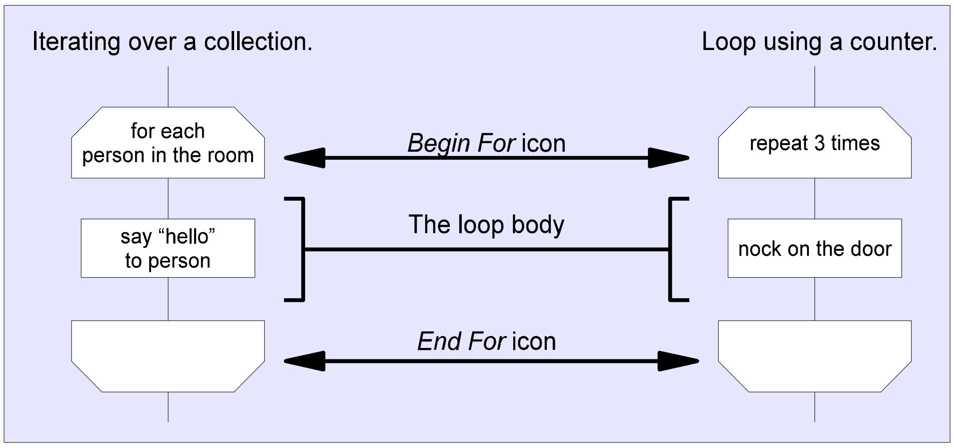
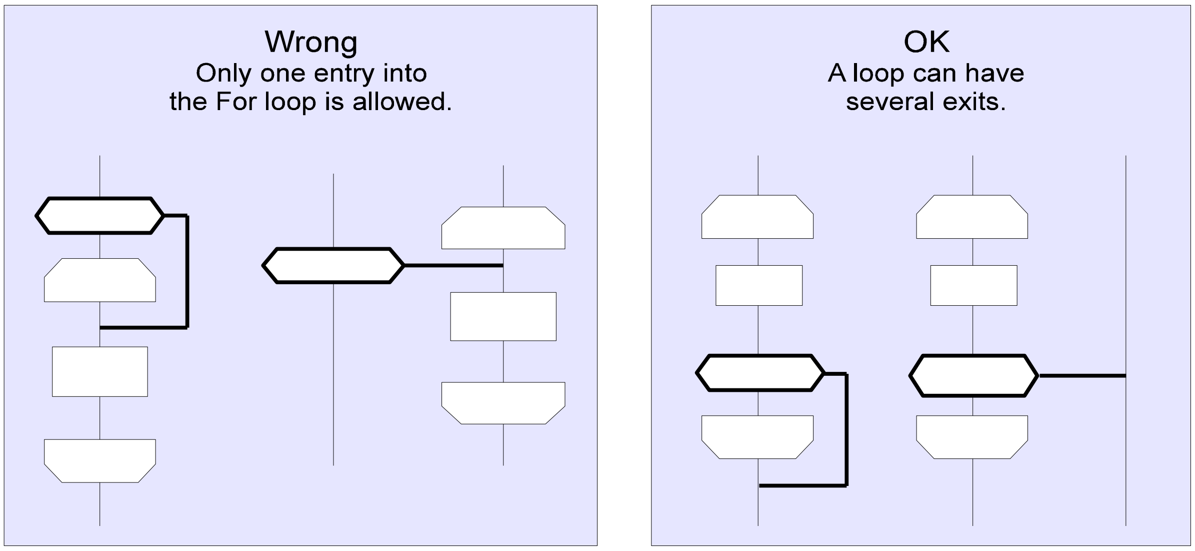


Fig 18. The For loop can have only one entry but many exits.


Wrong
Only one entry into the For loop is allowed.
OK
A loop can have several exits.

While, Do-Until and Hybrid Loops

Sometimes we need to perform an action many times as long as some condition holds. In this case a simple loop based on the *If* icon is a better choice. When should we prefer an *if* loop over a *For* loop?

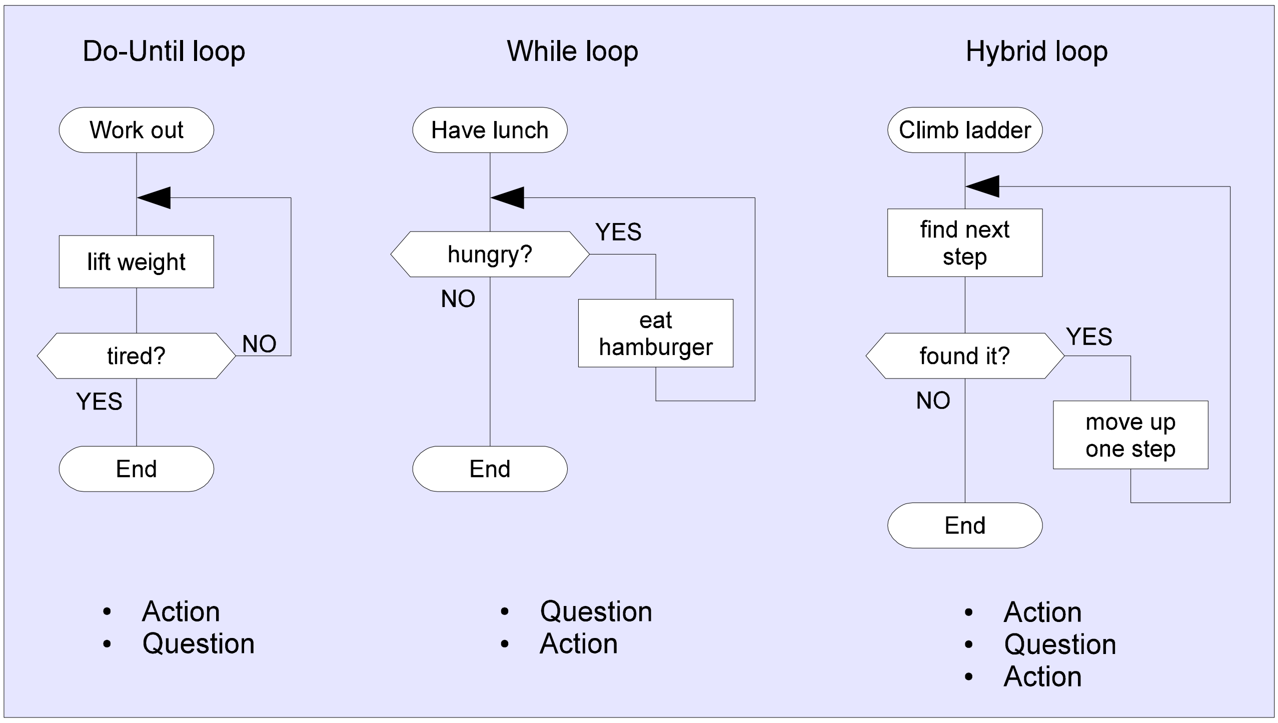
- When there is no explicit iteration over a collection or using a counter.
- When there is a repeating pair of two clearly defined steps: an *action* and a *check* for exit conditions.

The *If* icon can organize a loop in three ways:

1. **Question – Action.** This is similar to the *while* loop in programming languages.
2. **Action – Question.** This is similar to the *do-until* loop.
3. **Action – Question – Action.** This one is called the *hybrid* loop.

Note that a loop is the only situation when we direct a connecting line upwards. A line that is pointing up is such a rare exception that DRAKON ends that line with an arrow. All arrows inside a branch represent loops. All other lines do not have arrow heads because an excessive use of arrows adds unnecessary graphics complexity.

Fig 19. Loops based on the If icon.



Lines that go up are special in DRAGON and cannot carry icons on them.

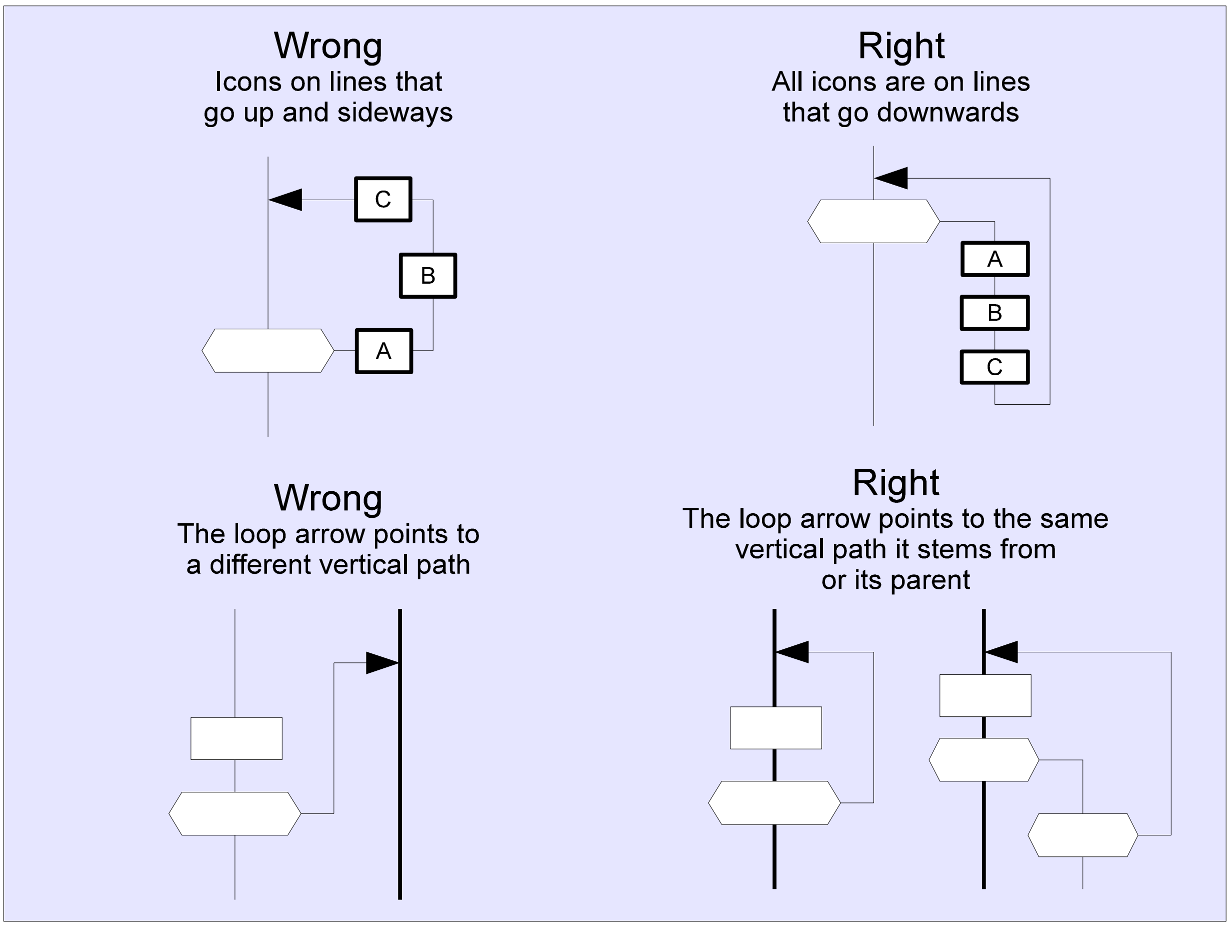
Rule: Never put any icon on a line that goes up or sideways.

This is a very important DRAKON principle – the next icon is below the current point of execution. Arrows just make the point of execution jump up above a certain icon.

Rule: Arrows never point to icons. Arrows point only to lines that go down.

This rule guarantees that for each icon, there is only one line that leads to it.

Fig 20. Wrong and right uses of the loop arrow.


Wrong
Icons on lines that go up and sideways
C
B
A
Right
All icons are on lines that go downwards
A
B
C
Wrong
The loop arrow points to a different vertical path
Right
The loop arrow points to the same vertical path it stems from or its parent

Nested Loops and Loops with Early Exits

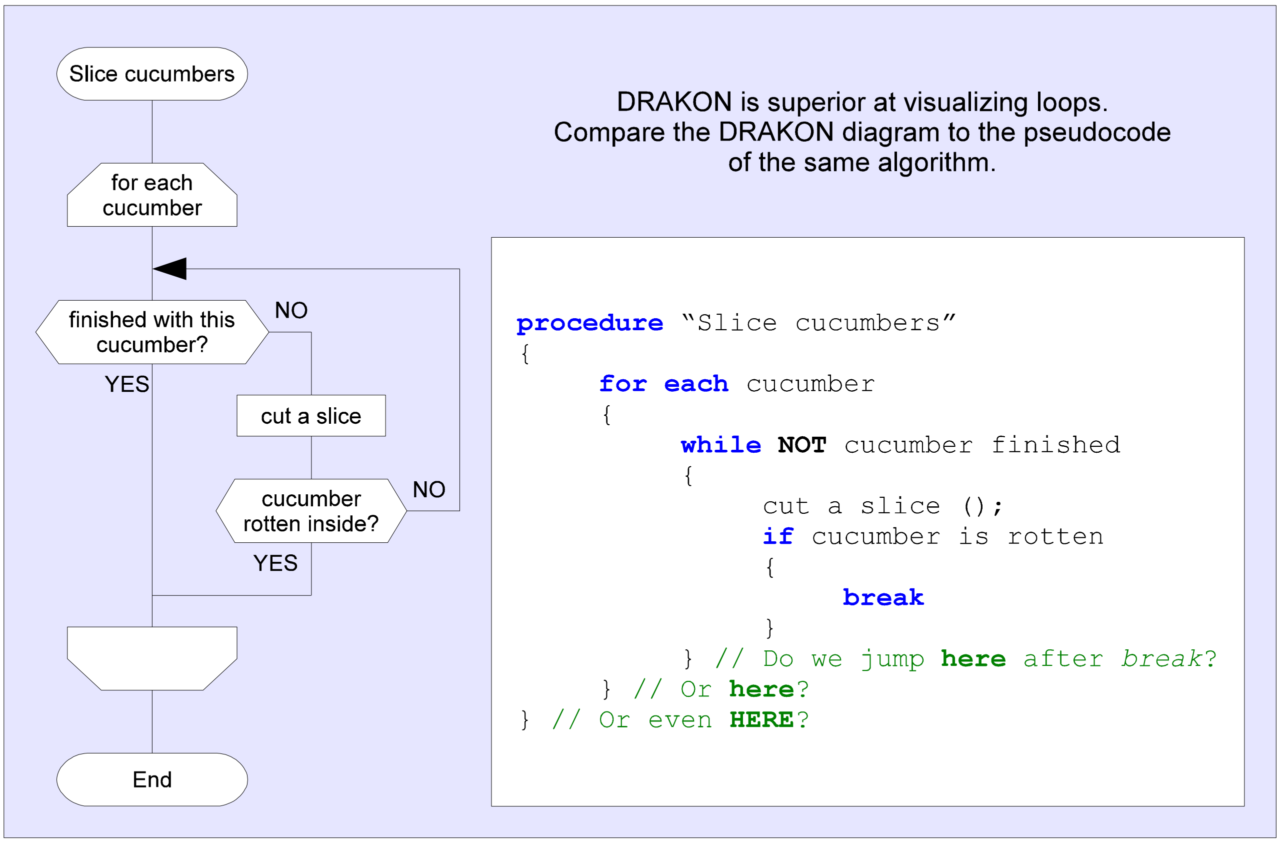
Programming loops with text is a strong habit, but it is a harmful habit. Traditional text-based ways of representing loops have several major drawbacks:

- The *for* and *while* programming constructs found in many languages force the loop condition to return *false* in order to quit. This is an arbitrary limitation that makes the programmer choose unintuitive identifiers like NotDone or introduce unnecessary *NOT* operators.
- It is hard to guess which of the loops will stop when we are exiting a nested loop.
- *If* statements increase indentation too and add chaos to even simple loops.

Non of the above problems apply to DRAKON.

DRAKON makes the program flow evident for any combination of loops and *if* statements. No matter how complex an algorithm is, it is always clear which icon will run next.

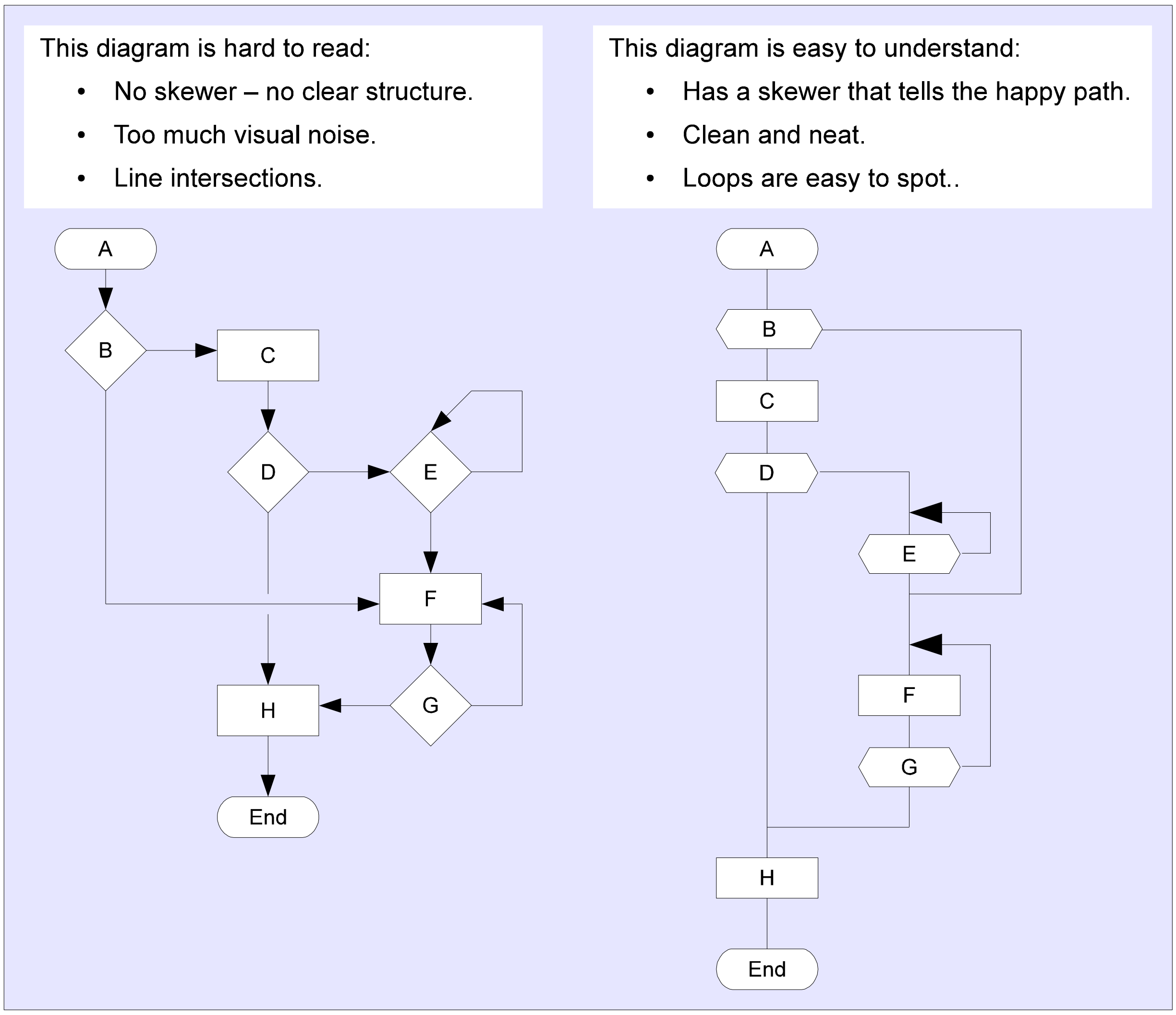
Fig 21. A nested loop with an early exit: DRAKON vs. pseudocode.



DRAKON is superior at visualizing loops.
Compare the DRAKON diagram to the pseudocode
of the same algorithm.

```
procedure "Slice cucumbers"
{
      for each cucumber
      {
            while NOT cucumber finished
            {
                  cut a slice ();
                  if cucumber is rotten
                  {
                        break
                  }
            } // Do we jump here after break?
      } // Or here?
} // Or even HERE?
```

Fig 22. Flowchart vs. DRAKON: the importance of ergonomics.



This diagram is hard to read:

- No skewer – no clear structure.
- Too much visual noise.
- Line intersections.

This diagram is easy to understand:

- Has a skewer that tells the happy path.
- Clean and neat.
- Loops are easy to spot..

Switch

The *If* icon works well when we have a question that can be answered “yes” or “no”. If the question has other answers we should use the *Switch* construct. The *Switch* construct consists of:

1. One *Select* icon that contains a question.
2. Two or more *Case* icons holding possible answers to that question.

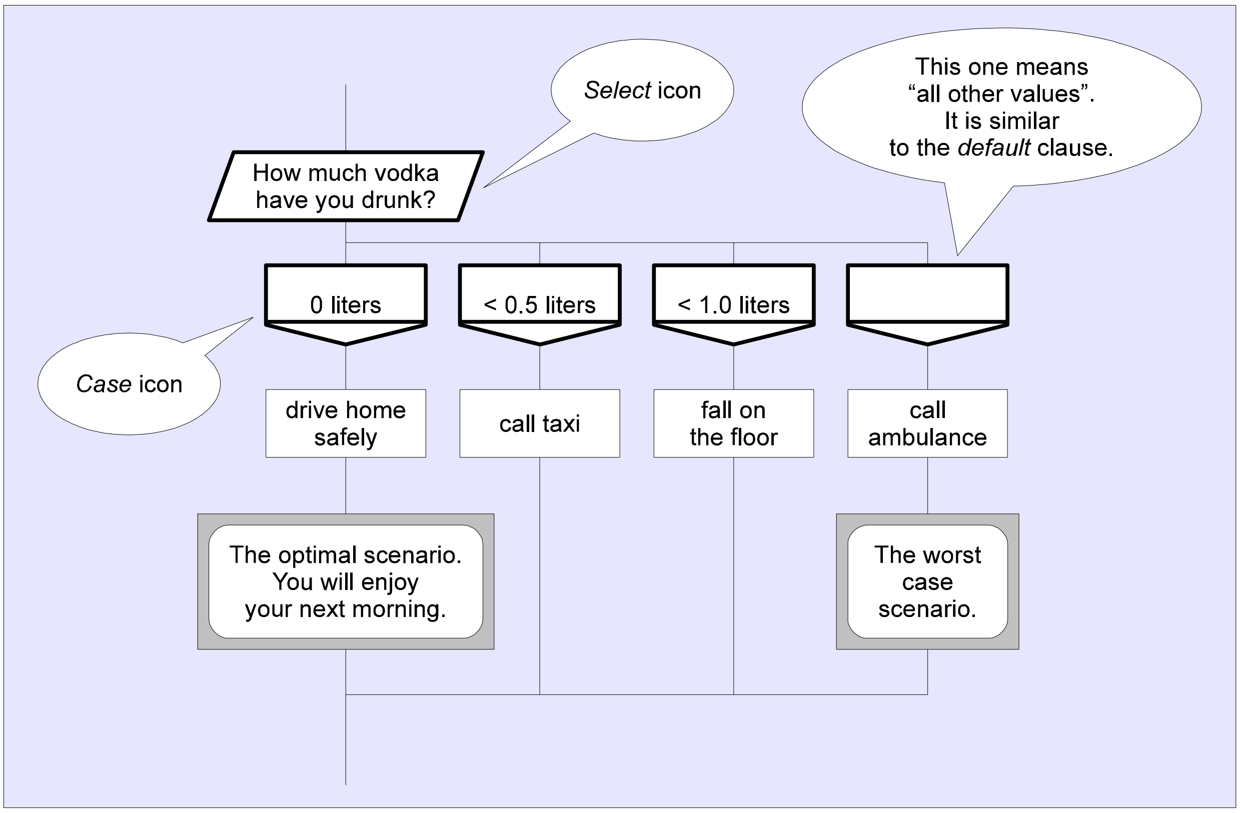
When placing answers in the *Case* icons, we should adhere to the basic DRAKON rules:

- The main route should go on the skewer.
- The further to the right, the worse it is.

If one answer is not much better than the others, we should find a criterion for arranging the *Case* paths in the ascending order from left to right.

The rightmost *Case* icon can be empty and have no value. This icon designates all other values and is similar to the *default* clause in the *switch* statement in some languages.

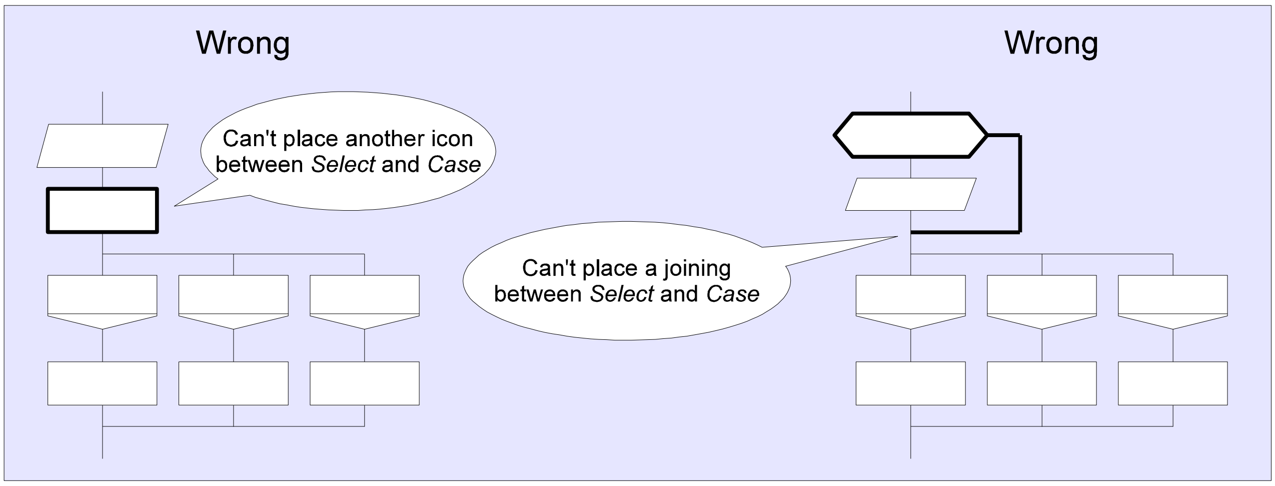
Fig 23. The Switch construct.


Select icon
This one means "all other values". It is similar to the default clause.
How much vodka have you drunk?
0 liters
< 0.5 liters
< 1.0 liters
Case icon
drive home safely
call taxi
fall on the floor
call ambulance
The optimal scenario. You will enjoy your next morning.
The worst case scenario.

The *Case* icons must immediately follow the *Select* icon.

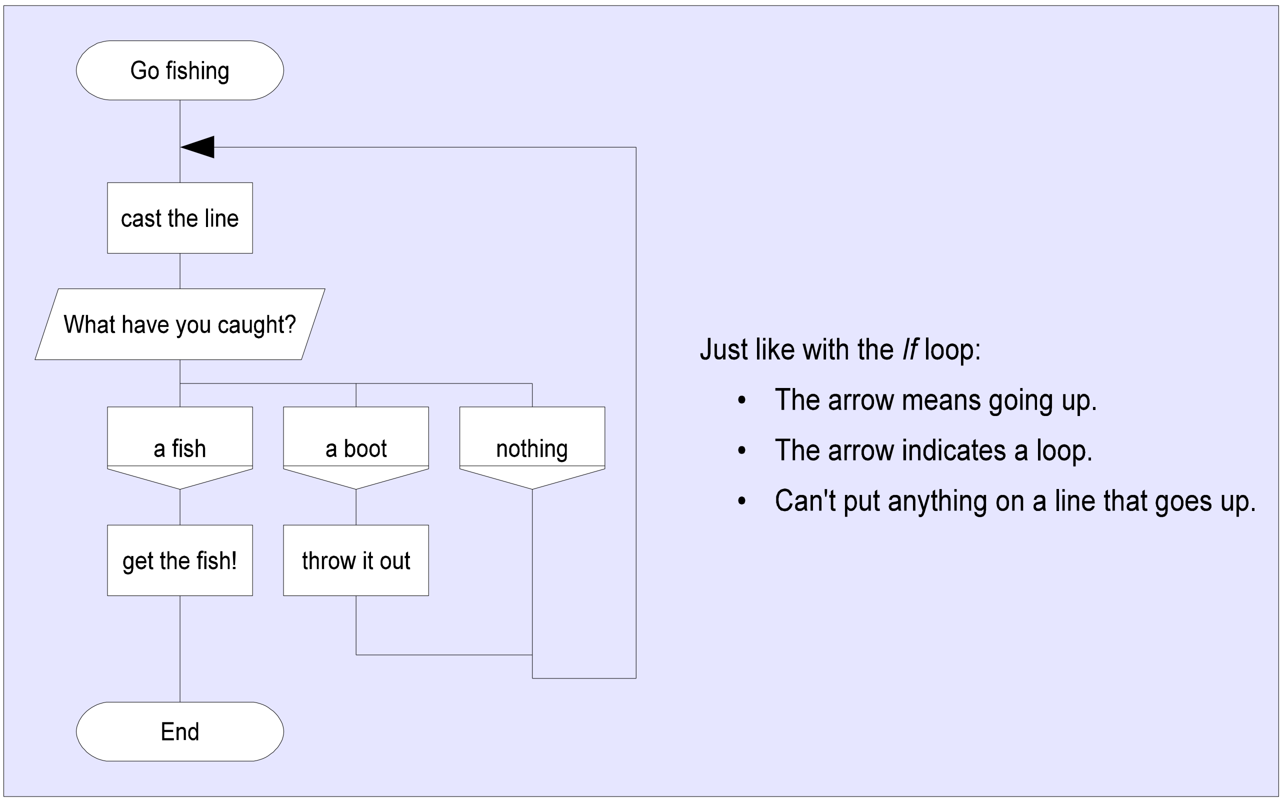
Rule: There should be no icons or joinings between the *Select* icon and the *Case* icons.

Fig 24. Switch errors.


Wrong
Can't place another icon between *Select* and *Case*
Wrong
Can't place a joining between *Select* and *Case*

The rightmost *Case* icons can lead to some place above the *Case* icon and form a cycle. This construct is called the *Switch loop*.

Fig 25. The Switch loop.


Go fishing
cast the line
What have you caught?
a fish
a boot
nothing
get the fish!
throw it out
End
Just like with the *If* loop:
• The arrow means going up.
• The arrow indicates a loop.
• Can't put anything on a line that goes up.

The Branch Loop

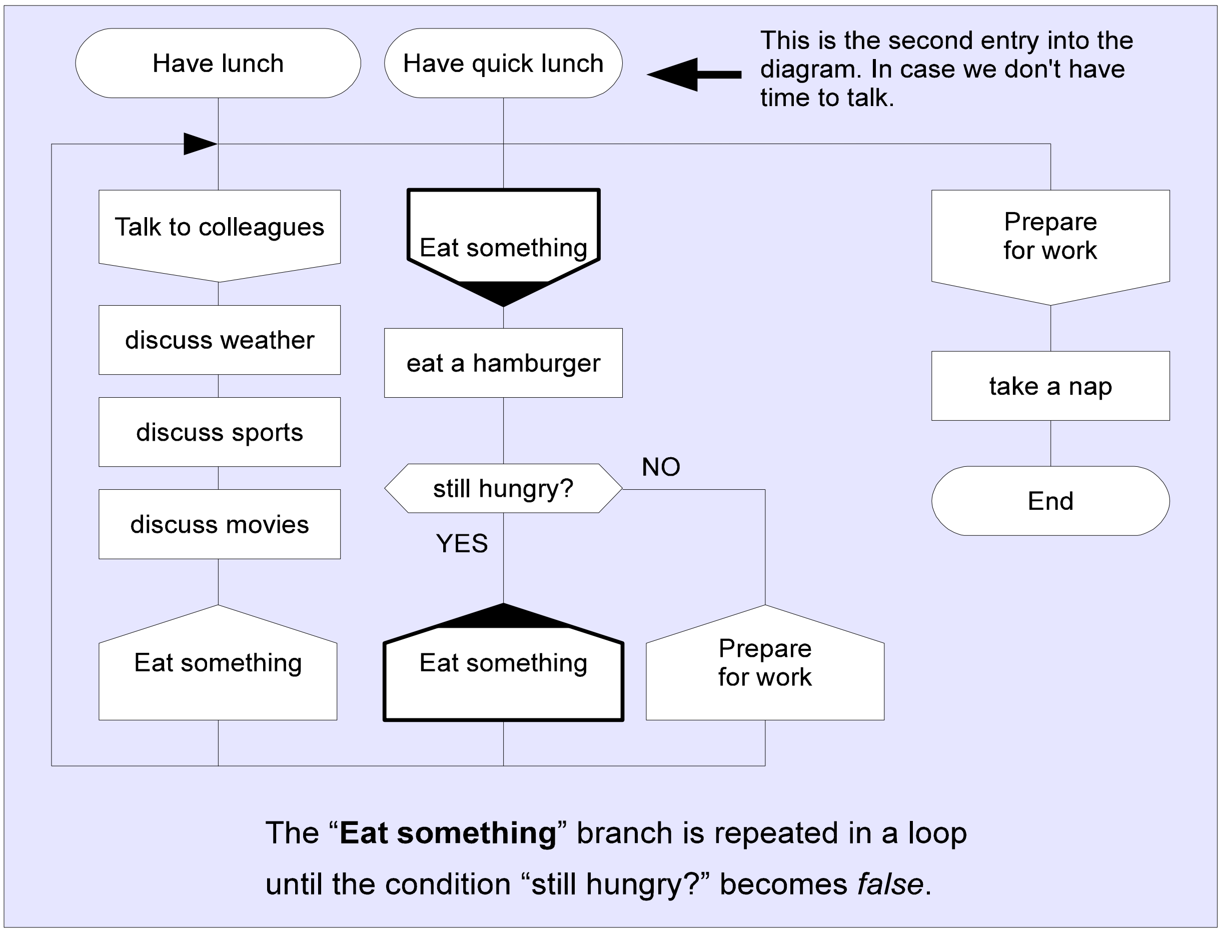
After a branch has finished working it transfers control to the next branch according to the *Address* icon. Technically, the *Address* icon can hold the name of any branch on the diagram, but the branch rule says that the next branch to call is the next branch to the right. There are exceptions:

1. Some branches may be skipped when an *If* or a *Switch* construct decides so.
2. The next branch can be the same branch or some branch to the left if the intention is to repeat some sequence of actions.

The latter case is called the *branch loop*. Just like any other loop, the branch loop can also have several exits.

One of the typical uses of the branch loop is to build a nested loop by embedding some other looping construct into a branch that is called repeatedly.

Fig 26. The branch loop.



The *Address* icon that causes the loop must have a special marking. That marking should also be placed on the branch that is the start of the loop body.

Logic Expressions

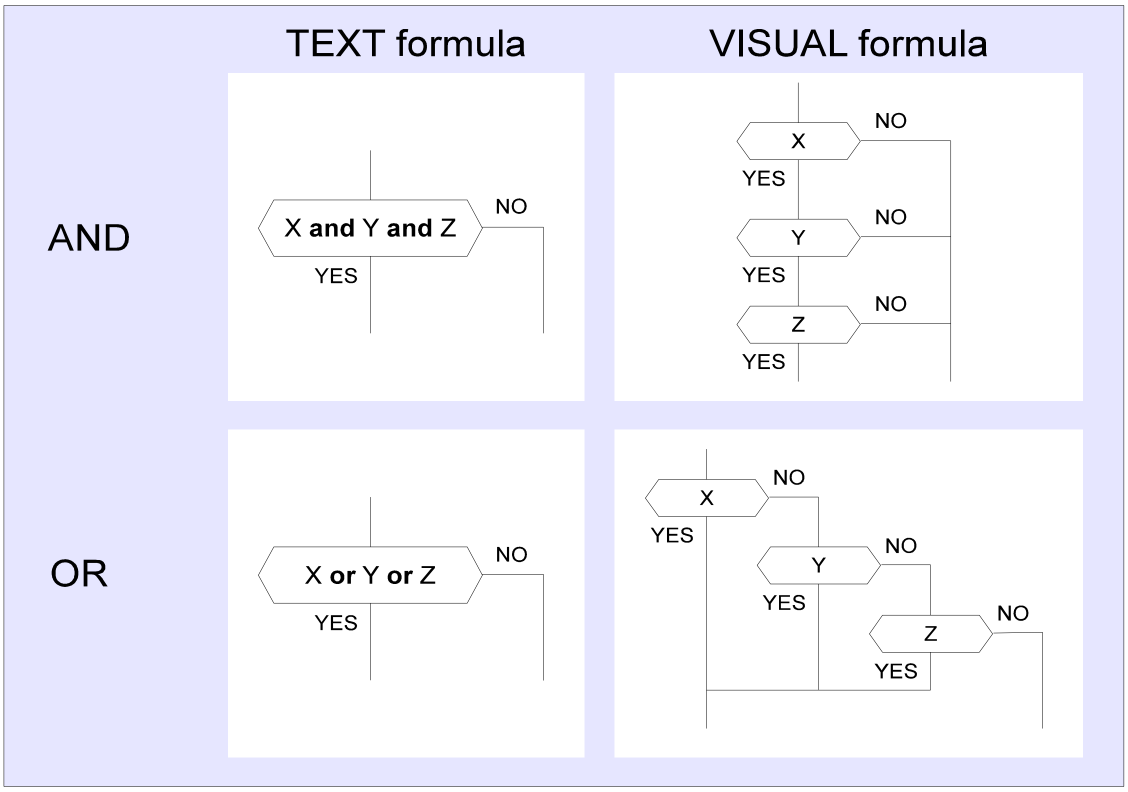
DRAKON offers two ways of representing logic expressions: textual and visual.

With the *textual* method, you write the whole expression inside the *If* icon, just like you do in the conventional programming languages. This method is not recommended.

The *visual* way has two steps:

1. Put each elementary operand of a complex logic expression into an individual *If* icon.
2. Connect the *If* icons in a way that ensures the equivalent order of evaluation of operands and end result.

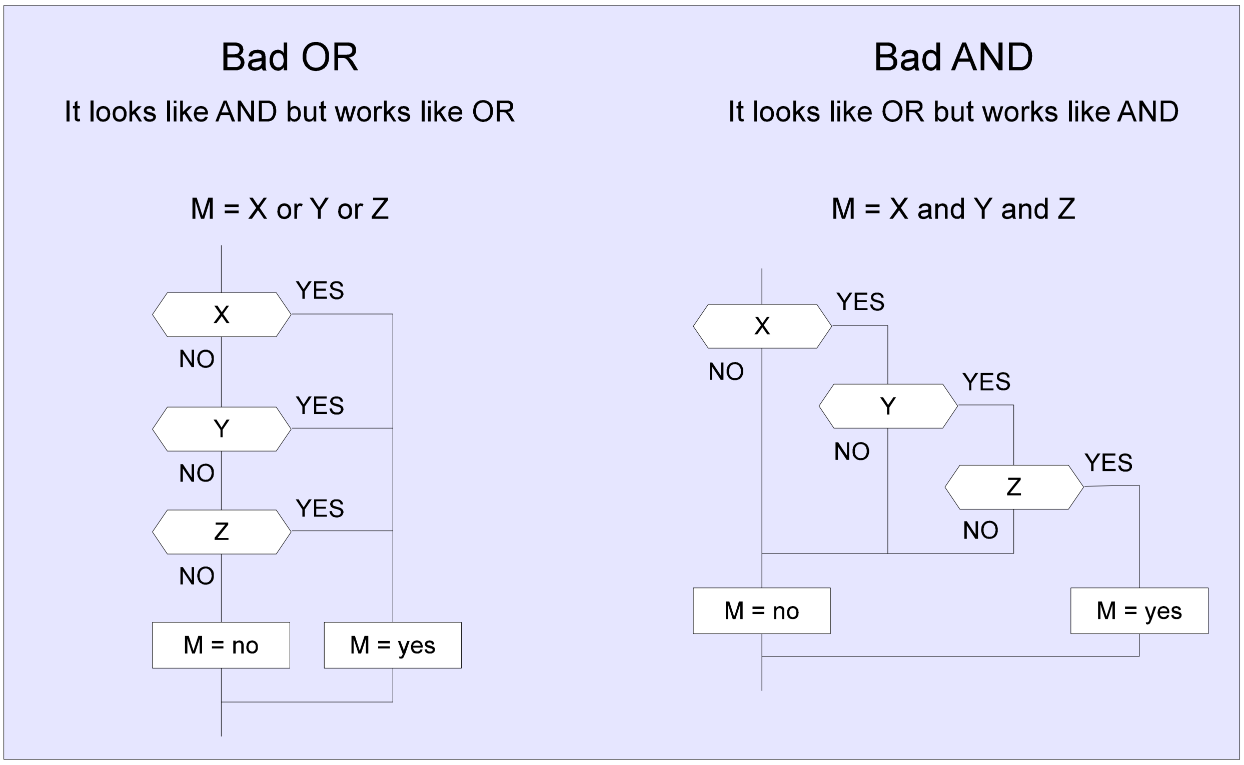
Fig 27. Textual and visual logic formulas.



These visual formulas form easily recognizable patterns which are beneficial to use.

Rule: For AND, put the *if* icons on the skewer. For OR, arrange the *if* icons as stair steps. The *yes* exits should be placed closer to the skewer whenever possible, while the *no* exits should go to the right. Going the other way around is allowed but not recommended because it breaks the patterns and misleads the reader.

Fig 28. Do not break the visual logic patterns.



The textual way of recording logic expressions is hard to understand and debug because it hides a big number of possible outcomes in a very short statement.

The so called *short-circuit evaluation* of logic expression adds more trouble to the problem. Many programming languages use this technique. It skips evaluation of the right operand of an expression when the outcome can be deduced from the left operand.

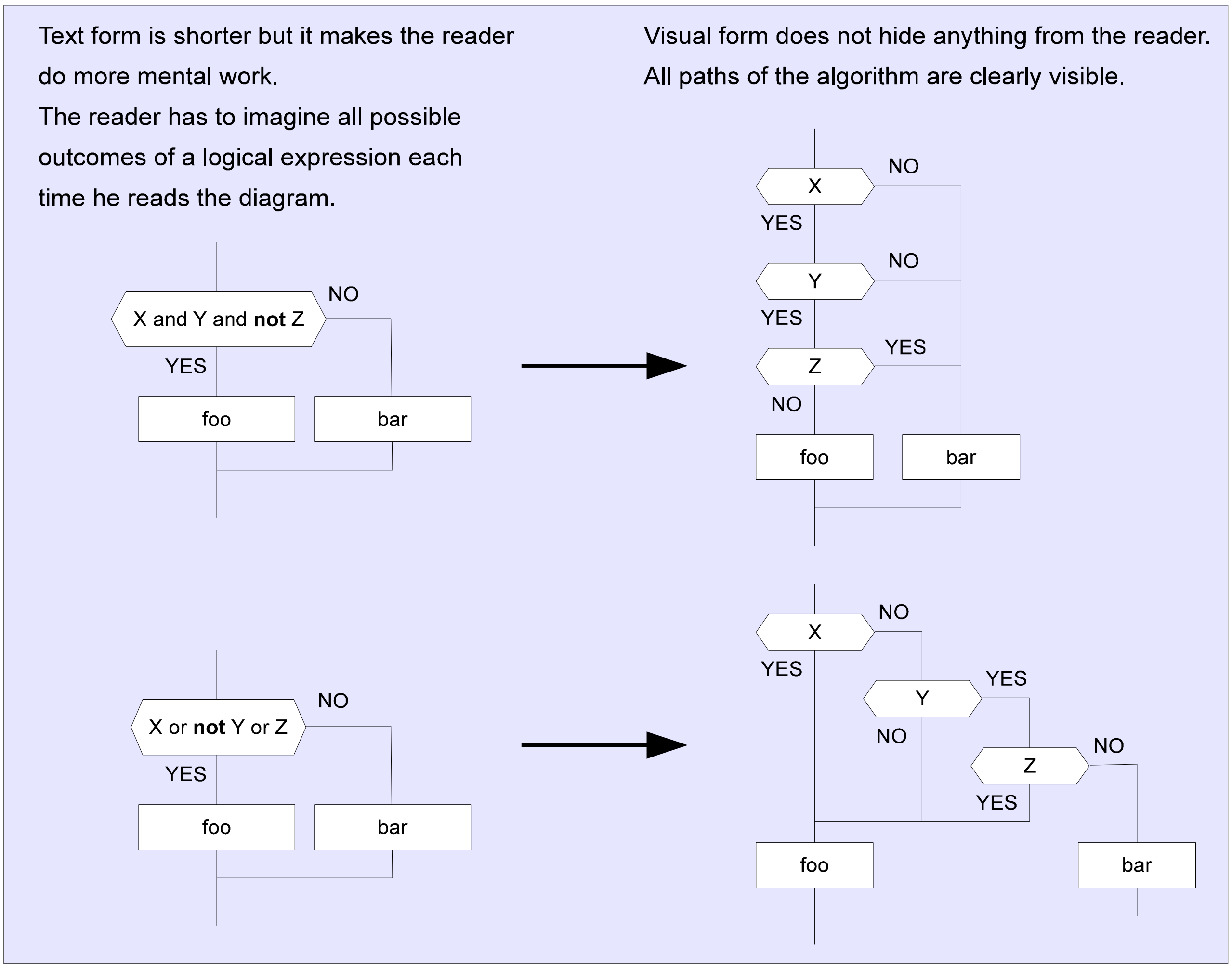
```
willBuy = LooksCool(gadget) and not TooExpensive(gadget)
```

In this example, the right operand (`not TooExpensive(gadget)`) does not need to be calculated when the gadget does not look cool – we will not buy it anyway.

Short-circuit evaluation adds implicit invisible paths to the algorithm.

DRAKON has a significant advantage because it shows all the possible paths of execution in an explicit and clear way.

Fig 29. The advantages of the visual logic formulas.

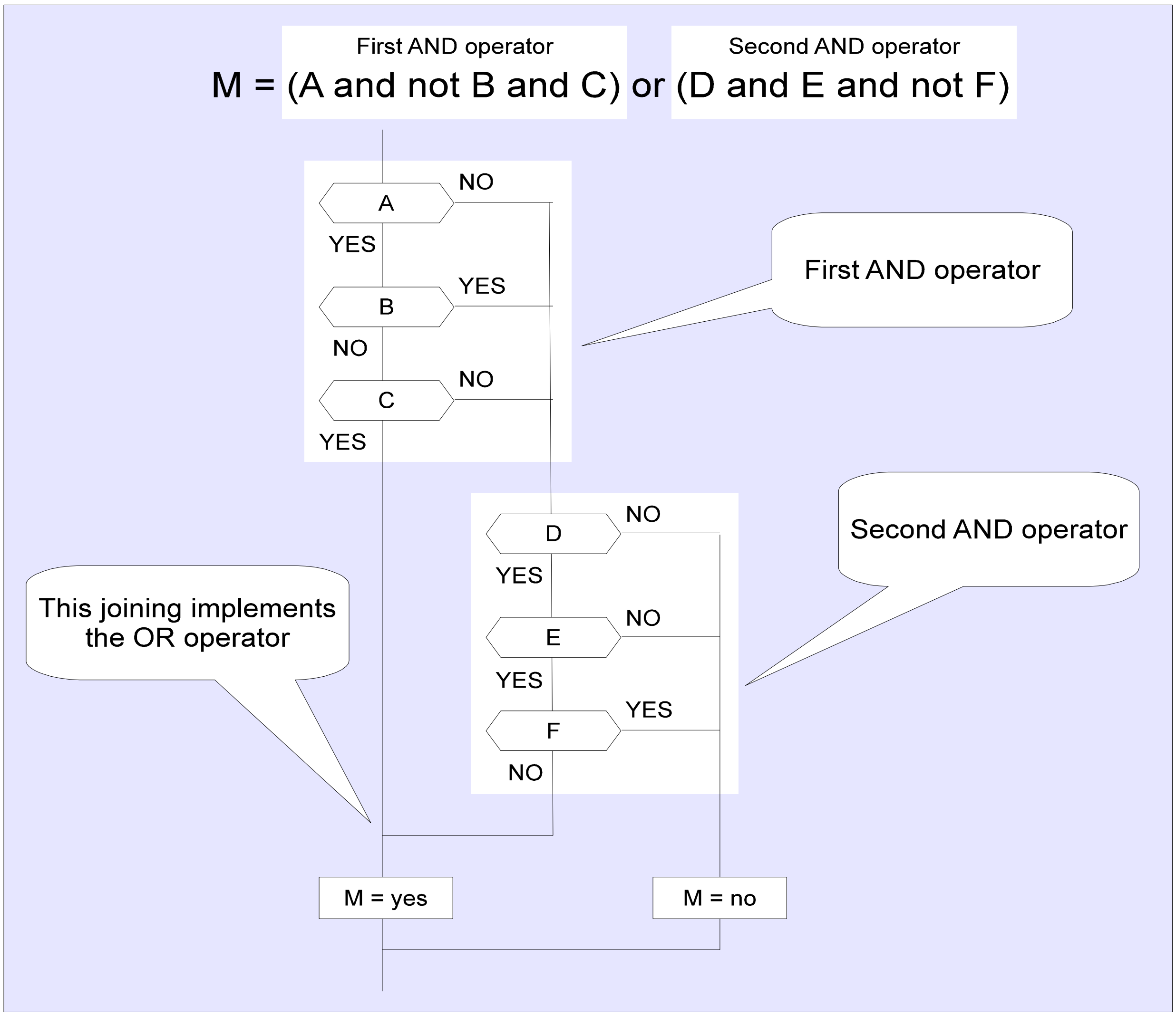


Text form is shorter but it makes the reader do more mental work.
The reader has to imagine all possible outcomes of a logical expression each time he reads the diagram.

Visual form does not hide anything from the reader.
All paths of the algorithm are clearly visible.

Note that there is no need for the NOT operator with the visual way of displaying logic. All the needed effects can be achieved by switching the *yes* and *no* exits from the *If* icons.

Fig 30. DRAKON explains complex logic formulas in an easy way.



Low-Level DRAKON and Real-Time Operators

The language that has been described so far is the high-level version of DRAKON. It is mostly oriented towards human readers and intended for documentation.

There is a more low-level version of DRAKON that has all the necessary details to be used for building programs. This version is called DRAKON-2. Icons in DRAKON-2 contain statements in a formal programming language like C++ or Java instead of free text. Assignments and method calls are done in the usual way, but flow control is delegated to the layout of the DRAKON diagram. During build time the diagram is transformed into a source file and then compiled.

DRAKON-2 has the so called *real-time* operators that add support for timers, concurrency, threading and input-output.

Summary

1. DRAKON is aimed at fast and easy understanding.
2. DRAKON helps the reader see the most important things first.
3. DRAKON pays great attention to detail in order to ensure clarity.
4. DRAKON is the best known way to represent loops.
5. DRAKON is superior at explaining logic formulas.
6. DRAKON has real-time extensions for real programming.

Sources and links

How to improve the work of your mind. V. Parondzhanov
http://drakon.pbworks.com/w/page/18205516/FrontPage

DRAKON. A short description. V. Parondzhanov.
http://narod.ru/disk/7290880000/0.%D0%94%D1%80%D0%B0%D0%BA%D0%BE%D0%BD%D0%9E%D0%BF%D0%B8%D1%81%D0%B0%D0%BD%D0%B8%D0%B5%D0%A0%D0%B5%D0%BA.rar.html

The history of DRAKON language.
http://www.transhumanism-russia.ru/content/view/331/116/

Buran and DRAKON programming language.
http://www.computerra.ru/readitorial/418507/

The History of Russian Shuttle (video).
http://video.mail.ru/mail/cherbatex1/12333/384.html

Buran spacecraft web site (in Russian and English).
http://buran.ru/