The Hierarchical Steps Pattern

Contents

Overview of Hierarchical Steps

A simple, common method of expressing a procedure uses a linear list of ordered steps: first do this, second do that, third do something else, etc. This basic method of describing procedures occurs across many different domains of human and computer activity.

Hierarchical Steps uses an ordered tree structure to express the steps of a procedure: one root node for a step that describes the entire procedure, with any number of ordered child nodes or steps, each one of which may have ordered children as well, and so on to the leaf nodes or steps of the tree. With Hierarchical steps, a step begins before any of the descendant steps of that step begin, and ends after completion of all the descendant steps or following failure in any of the descendant steps. This approach lends structure to the set of steps, including which steps are larger in scope or smaller, and which steps contain, or cover the scope of, other steps in the hierarchy.

This paper describes the utility and easy naturalness of using hierarchical steps to express a procedure.

Problem and Motivation in Context

Given a procedure where the *how* of carrying out the procedure is important, an ordered list of steps seems natural, given how steps are verbally delivered between people or the long history of logging discrete statements in software and systems with no higher-level structure. However, the simple ordered list is limiting in clarity and usefulness.

A list of steps presents the steps as if they were all equally important, of equal scope or impact, but this rarely represents the reality.

One way of making an ordered list easier to follow is to describe multiple steps and details within each numbered step of the ordered list. This corresponds to a two-level hierarchy, with multiple child steps written out as the body of each numbered top-level step, but when expressed with long sections of prose, this can make for confusing reading. Such steps can get long and/or complicated, and written prose does not always translate directly into discrete, ordered steps for a procedure.

The problem is to describe some procedure in a natural, executable, repeatable way, such that:

- The large-scope steps are clear, for efficient communication in commonly understood terms and for thinking about and prioritizing the issue or issues addressed by the steps
- Make all the details of the steps accessible as needed, without making the big, important steps confusing, verbose, or complex, or risk the reader's misunderstanding of what was intended
- Limit the change impact or risk of adding, removing, or changing a step, especially the detail-oriented technology-facing steps

The Solution and Result

The solution is to represent the steps of a procedure in an ordered tree, i.e., a hierarchy. If there is one node at the root of the tree (typically, this is graphically represented at the top), that one node represents the entire procedure at the highest or most general level, the ordered child nodes of the root represent the biggest steps that comprise the root node, and if one drills down to the leaf nodes, one finds the smallest-scope (but potentially quite important!) detail steps of the procedure. Each step or node begins before any descendant step, and completes after all the descendant steps have completed or after a failure in a descendant step.

If details are missing, add more nodes, usually as leaf nodes.

The result is a structured hierarchy that represents the entire procedure, with as many fine details and as finely detailed as needed, but the overall purpose and execution of the procedure are clear.

The ordered tree, representing a hierarchy, is how procedures are communicated and executed naturally.

More advantages to using a hierarchy for communicating a procedure are discussed in later sections.

Examples

Flying from Seattle to Brussels

I'm flying to Brussels. If it comes up in conversation, that's how it's expressed: "flying from Seattle to Brussels."

A linear list of steps titled “Fly from Seattle to Brussels” might look like this:

1. Drive to Parking lot near airport
2. Leave car
3. Leave charging fob (because it’s an electric car)
4. Get ticket for car
5. Take shuttle to departure terminal
6. Find TSA security
7. Stand in line
8. …

Gets boring, doesn’t it? There’s not much structure to guide this presentation, just a long ordered list of steps. Finding the context for any individual step needs examination of an arbitrary number of preceding steps. The overall purpose of the steps is in a title, but the title is a piece of information disconnected from the steps themselves. Adding more information with more steps makes the list longer, more prone to change, and therefore less useful to compare to other lists or even other iterations of the same list.

By contrast, let’s put this process into a hierarchy. To start simply, look at just the root node, which corresponds to the title, purpose, or intention of the whole process, and the child nodes of that node, which correspond to items we might use in conversation to explain the trip:

1. Fly from Seattle to Brussels
    a. Travel to Seattle airport
    b. Fly to London
    c. Fly to Brussels
    d. Go to hotel

The title is no longer disconnected because it’s the root node. It’s no longer boring because this structure allows a much more succinct and germane presentation of the most important steps. The context of any step is trivially easy to find; just look to the parent node of the step in the hierarchy. More details can be added as child nodes, which does not perturb the presentation at all for the higher-level nodes.

Describing the trip more fully, at a detail level analogous to the long boring list above, gives

1. Fly from Seattle to Brussels
    a. Travel to Seattle airport
        i. Drive to Parking lot near airport
        ii. Leave car at parking lot
            1. Find attendant at lot
            2. Leave charging fob
            3. Get ticket for car
        iii. Take shuttle to departure terminal
    b. Fly to London
        i. Do TSA security
            1. Find TSA security
            2. Stand in line

3. …

Here are some advantages that appear with the hierarchical structure, as compared to the linear list:

- The overall topic is easy to find in connection with the rest of the hierarchy, because it's the root node.
- The structure is expressed in the hierarchy.
- The context for any node is simply the parent of that node.
- Dependencies or details for any node are simply the descendants of that node.
- Additional details can be added as leaf nodes, or even new nodes with children, with no changes to most of the structure and relationships.
- Nodes can be changed with no disruption to parent nodes or independent parts of the structure.
- Complexity (and risk of failure at a step) are managed by the structure of the hierarchy.
- Peer, i.e., sibling nodes have similar scope.

By contrast, with the long list of ordered steps, scope or impact of a change is not expressed by the structure other than the order of the steps. For example, if a step is added, deleted, or changed, the question of which other steps are impacted can only be answered tentatively by examining the nearest steps first and then the ones further removed in the list. There is no structure at all to express which steps are more or less important than which other ones. With a list of ordered steps, adding more detail tends to change the whole list.

Installing a Dishwasher

In conversation, one might say "I installed a new dishwasher." If that statement were the root node of a hierarchy, the child nodes would include "Remove the old one and clean the space," "supply electrical power," "hook up water supply," "hook up drain," and "test dishwasher and installation." Each of those nodes has many more details as well, which, with a hierarchy, can go into child nodes.

The hierarchy of steps is partly represented in the installation manual. The creators of the installation manual knew well that if the instructions were just in a long ordered list of steps, it would be difficult to follow, so the steps are grouped into sections. Translated to a hierarchy, the manual represents three levels in the hierarchy: the root node, at the top level, is the overall purpose of the instructions, "Installing the Dishwasher." The sections form child nodes of the root, and the steps listed in the sections are child nodes of the section nodes.

The installation manual also has break-out diagrams with detail. These could be represented by child nodes of the step nodes, which would represent a fourth level of the hierarchy.

Cooking from a Recipe

Suppose someone asks for your lasagna recipe. If "Lasagna" is the root (highest) node, then the child nodes might be 1. The protein, 2. The cheese sauce, 3. The pasta, 4. How to put it all together and 5. Bake.

Each of the child nodes has of course many more details expressed in the lower detail nodes of the recipe hierarchy. For example, step 3 "The pasta" needs details on a) what kind or kinds of pasta works and b) how to cook this separately.

Dividing up the recipe into parts, expressed here as nodes, makes it easier to do substitutions. For example, suppose one wants to make a vegetable lasagna; just change the child nodes of the “The protein” node from choosing and cooking a meat base to doing the corresponding with a vegetable mix. Unrelated nodes do not change.

Related Methods

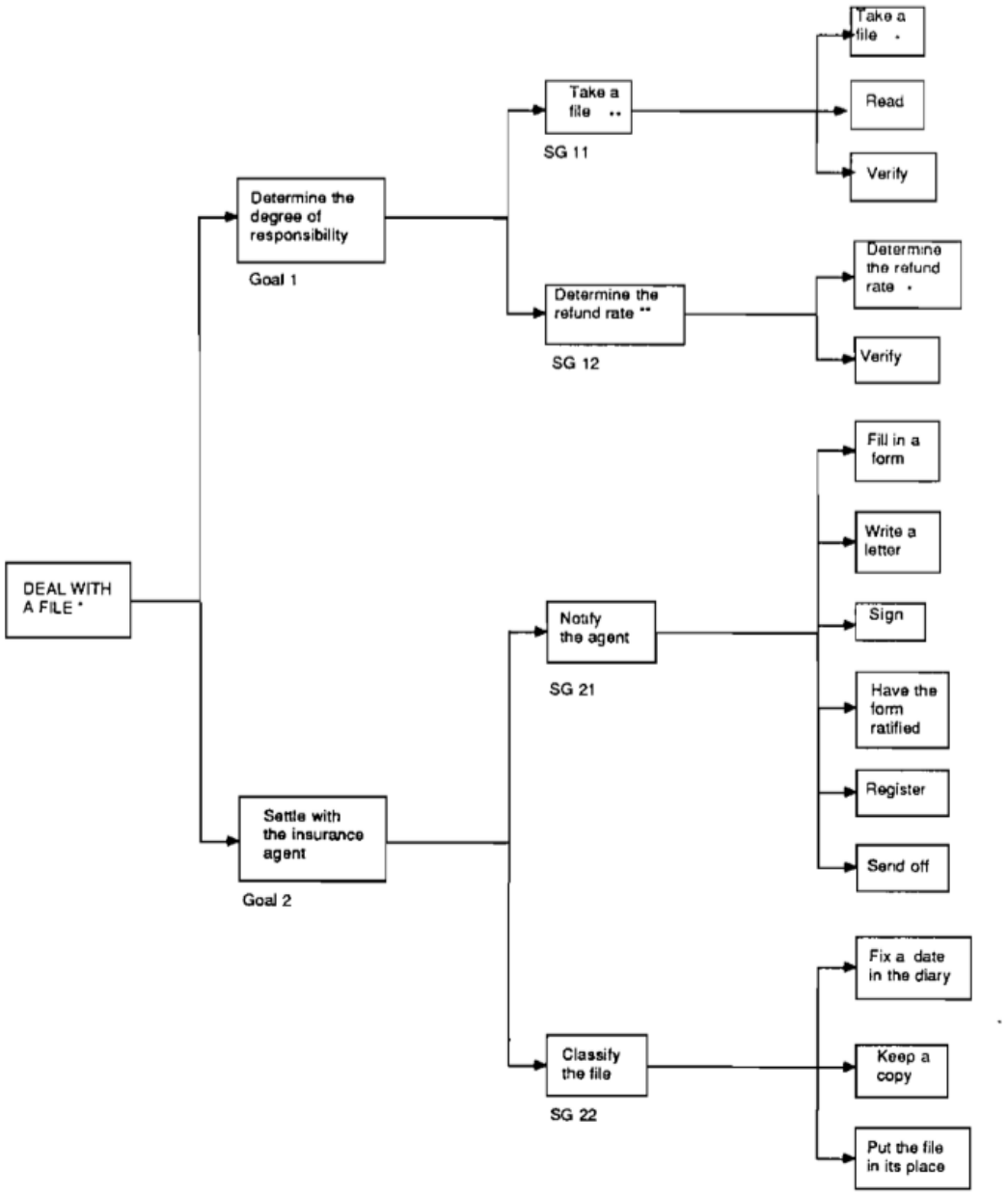
Hierarchical Task Analysis

With the goal to “… improve the design of the man-computer interface …” Suzanne Sebillotte laid out how steps of achieving a task can be optimally arranged in a hierarchy, in her paper “Hierarchical planning as method for task analysis: the example of office task analysis.”[1]

Sebillotte develops a model of four levels to describe the task of working with an insurance claim at an insurance company, here labelled at the top level of the hierarchy “DEAL WITH A FILE.”[2] Note that to work with her available presentation space, she depicts the “high” levels of the hierarchy towards the left, with "low" levels towards the right.

[1] Suzanne Sebillotte, “Hierarchical planning as method for task analysis: the example of office task analysis,” Behaviour & Information Technology, 7:3, 275-293, DOI: 10.1080/01449298808901878

[2] Sebillotte notes that the labels for the nodes are translated from the French, and correspond to the words of the person describing his or her procedure in accomplishing the task



Sebillotte describes her process in discovering the hierarchy of task steps:

> *At the top of the hierarchy, the task may be described in an abstract way (its name or the objective of the task). The lowest level of the hierarchy must be understood as 'the lowest verbalizable level'; subjects are unable to provide more details about the procedure used, or even do not understand the question.* [3]

[3] Sebillotte, p. 9

Sebillotte's descriptions of how she creates steps at the highest and lowest levels of the hierarchy anticipate the Hierarchical Steps pattern, and clearly show the distinct values of the higher and the lower levels of the hierarchy as well as the value in the intra-step relationships represented by the hierarchy.

Concur Task Trees

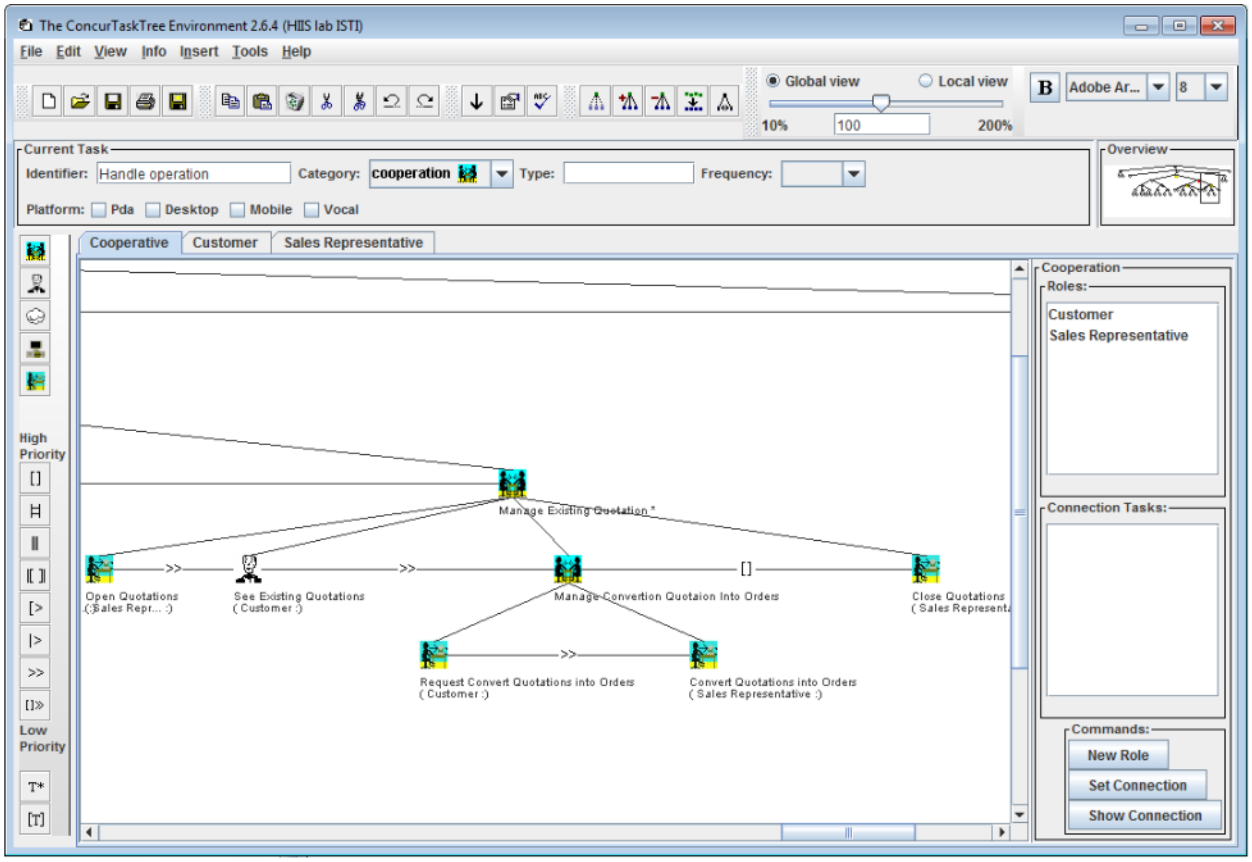
Concur Task Trees (CTT) is a graphical notation for representing a hierarchical task model, i.e. "… the logical activities that an application should support to reach users' goals."[4]

The naturalness of hierarchical structure is noted in detail, along with some other advantages, on a CTT summary published on the Human Performance repository of the SESAR Joint Undertaking for European Air Traffic Management infrastructure:

> *Hierarchical structure, a hierarchical structure is something very intuitive, in fact often when people have to solve a problem tend often to decompose it into smaller problems still maintaining the relationships among the smaller parts of the solution; the hierarchical structure of this specification has two advantages: it provides a large range of granularity allowing large and small task structures to be reused, it enables reusable task structures to be defined at both a low and a high semantic level.* [5]

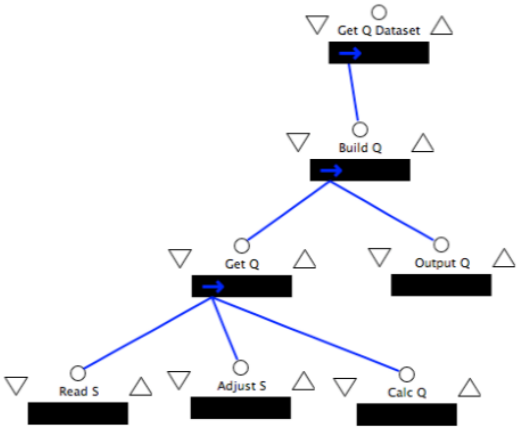
CTT is developed by the HIIS Laboratory, which also provide a software tool "ConcurTaskTrees Environment" to aid in developing CTT models.[6] The tool site provides the following graphical representation of the tool and a CTT project:

[4] Wikipedia, https://en.wikipedia.org/wiki/ConcurTaskTrees, referenced July 31, 2016
[5] Human Performance (HP) repository, https://www.eurocontrol.int/ehp/?q=node/1617, referenced July 31st 2016
[6] The tool is available here http://giove.cnuce.cnr.it/ctte.html



Little-JIL

Little-JIL is a coordination language, used to define processes that coordinate activities of multiple agents.[7] The following figure shows an example of a simple coordination diagram:[8]



[7] Barbara Lerner, "Getting Started with Little-JIL, Case Study: Measuring Stream Discharge," Mount Holyoke College, May 2010, page 1

[8] Ibid, page 6

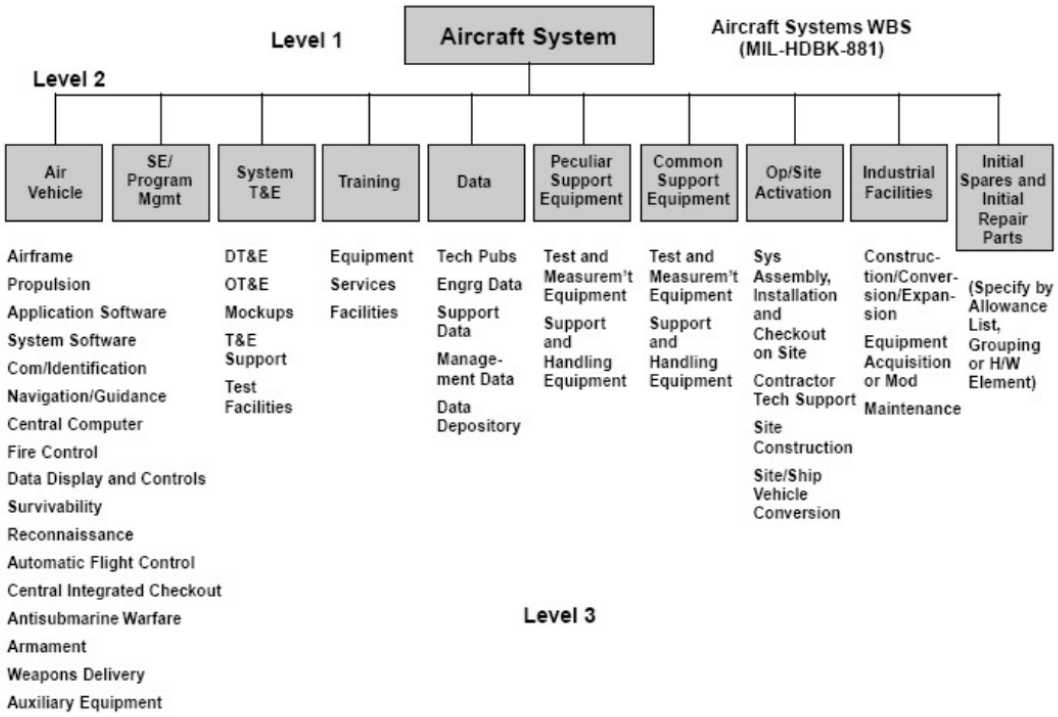
Note that the hierarchical structure provides context as part of the structure. For example, the node “Get Q” provides the context for the three child nodes, “Read S,” “Adjust S,” and “Calc Q.” The three child nodes in turn supply meaning and detail to “Get Q.”

The hierarchy is easier and simpler to read and navigate than a long linear list, and if more detail is needed, for example for the “Calc Q” node, it can be added as child node or nodes to “Calc Q” and the rest of the hierarchy is not impacted.

Business Process Modelling

Top-down modelling, a kind of business process modelling, also uses hierarchical relationships.

A work breakdown structure in project management and systems engineering is a deliverable-oriented decomposition of a project into smaller components, which textually or graphically represents a hierarchy.[9] The graphic below represents an application of this towards military aircraft, showing the first three levels of a typical aircraft system.[10]



The work breakdown structure depicted here shows, just as Sebillotte’s hierarchical planning diagram does, the value of the higher-level nodes in depicting context for the lower-level nodes, and in the relationships described by the structure. For example, in this diagram, “Airframe" is a child of the “Air Vehicle" node, not the “Training” node.

Applications for Software

Representing the Procedure of a Stack-Based Language

[9] Wikipedia, https://en.wikipedia.org/wiki/Work_breakdown_structure, referenced July 8th 2016
[10] ibid

Given a procedural view of a stack-based language, e.g., C, C# or Java, the sequence of calls in executing a static piece of code naturally forms an ordered tree hierarchy. For example, given this C# code:

```
class Class
{
    public void BaseMethod()
    {
        MethodA();
        MethodB();
        MethodD();
    }
    private void MethodA()
    {
        MethodD();
    }
    private void MethodB()
    {
        MethodC();
    }
    private void MethodC()
    {
        MethodD();
    }
    private void MethodD()
    {
        // DoSomethingHere
    }
}
```

The procedure maps to this ordered tree hierarchy, represented in XML:

```
<?xml version="1.0" encoding="utf-8" ?>
<BaseMethod>
  <MethodA>
    <MethodD>
      <DoSomethingHere/>
    </MethodD>
  </MethodA>
  <MethodB>
    <MethodC>
      <MethodD>
        <DoSomethingHere/>
      </MethodD>
    </MethodC>
  </MethodB>
  <MethodD>
    <DoSomethingHere/>
  </MethodD>
</BaseMethod>
```

Assuming that the C# code has precedence and the XML represents the code, one would use reflection to build the hierarchy in XML from the code at runtime. In case the C# code is dynamic or generated at runtime, reflection can still build the corresponding hierarchy as the code is executed.

Presenting a Repeated Procedure for Performance and Behavioral Analysis

The previous section shows how to represent a fixed multi-step procedure, implemented with a stack-based language, as a hierarchy.

A hierarchy also has advantages for *analysis* of a fixed procedure, as compared to merely describing the procedure as a long ordered list of steps, in three areas: failure impact, performance, and handling occasional procedure changes.

If there is a failure at one step, other steps might also logically fail for the same root cause, while others are blocked from execution. A hierarchical representation of the relationships between these steps supports clear and actionable relationships that show which steps fail, versus which ones are blocked: from the root failure step, which is as close to root cause of the failure represented by the steps documented in the hierarchy, all ancestor steps (i.e., the parent step, the grandparent step, the great-grandparent step, etc.) also fail. All peer steps, and any descendants of those peer steps, that follow the failed steps are "blocked" because they are not executed at all in this failure case.

Any failure is thus naturally communicated up to the top level of the hierarchy; any failure in the procedure hierarchy fails the entire procedure.[11]

By contrast, a linear list of steps does not support any failure relationships because there is no relationship, other than the sequence itself, between neighboring steps. A failed step can only communicate failure to the overall procedure, and following steps are blocked.

If performance is measured for each step in a hierarchy by time span for execution (e.g., in milliseconds), the component or child steps of any step that has children show clearly how performance is broken down, showing changes in performance in greatest detail at the leaf steps, and allowing focus on where performance issues might benefit from attention. By contrast, a linear list of steps does not support drilling down to root cause of a performance change, or show as precisely the best focus for improving performance.

If there is an occasional externally-directed change to the steps of a procedure, the hierarchical structure shows clearly the scope of the impact of the change through parent, peer and child steps to the steps that are added, removed or changed. When a step changes, all descendants of that step are likely changed as well, in name, order, hierarchy etc. but no changes to ancestor steps are implied. Other changes to the hierarchy may be directed externally, so the entity directing the change to the procedure is responsible for making changes elsewhere in the hierarchy that are neither ancestors nor descendants of the changed node, in the case that this is necessary. In this way, managing any changes to the procedure is easier, simpler, and lower-risk if the procedure is represented as a hierarchy, as compared to the more traditional ordered list.

A linear list of steps does not support grouping or steps of varying scope, so all changes must be handled at the same level of scope, and any resulting risk from the change is at the same level of scope as well.

The implications of any procedure changes to performance and reliability are therefore more clear in case a hierarchy is used to represent the procedure, as opposed to an ordered list of procedure steps.

Automating Software as part of Quality Automation

[11] This is how it's done in sample code available on the site http://MetaAutomation.net.

When programmatically making software do stuff, i.e., automating it, and measuring what the software is doing in response, the details of the steps driving the software are very important because the measurement result might depend on the identity and order of any or all of those small steps, especially in case of failure of a step or significant performance change. A missing step, or step that is done differently than expected, will likely influence the result. Even more so, if the software changes during product development, the steps themselves might change as well, which becomes very important information when doing analysis on functional or performance aspects of the software quality. Mostly, however, the order of the steps and relationships between the steps do not change for this usage scenario.

The steps that are used to drive (or use, from the end-user perspective) software naturally fall into a hierarchy. There are larger-scope steps that describe the higher-level purpose, up to the root node, and often very small-scale steps that describe exactly how the software is driven in the smallest and non-divisible, i.e., atomic steps. Whether the automation code is written statically or dynamically, the code methods could represent nodes in the hierarchy of steps. To make the code simpler and easier to read (and, to simplify management of the scope of automatic variables) multiple steps and child steps can be written into a method using anonymous delegates.[12]

For steps that are repeated, e.g., for a well-defined procedure that seldom changes, the structure makes it possible to identify the expected steps at runtime and, in case of check failure, which steps are failed and which are blocked. This in turn makes it clear what quality measurements are blocked, so the uncertainty in the quality of the software system, i.e., the product operations that are un-measured, is clearly laid out.[13]

How to Make a Hierarchy for a Procedure

Start by defining the highest level, the root of the hierarchy, with a node that defines the procedure. For example, this node could have the name “Fly from Seattle to Brussels.”

Define the child steps (nodes) of the root. Each child step represents a distinct part of achieving the goal, objective, or activity described in the root node of the hierarchy; these child steps of the root are sibling steps to each other. Define the steps so they’re distinct and the purpose of each is clear, but do not put further details into the steps themselves. All child steps complete (or end with failure, or are blocked due to an earlier failure) before the root step completes (or, ends with failure).

Details for each step go into child steps, so create child steps as needed, with peer steps and child steps as those steps as needed. Make each step as simple as possible, while still being meaningful and identifiable.

For defining steps that are expressed in computer code, there may be no need to create steps so finely detailed that any step has zero probability of failure, so combine statements together if it makes conceptual sense to group them. At the same time, ensure that no leaf step contains more than one potential point of failure.

---

[12] For examples of this, please see sample code available on the site http://MetaAutomation.net.
[13] Ibid

Go as fine, so make child steps as far, as is appropriate for your decision-making domain. The granularity of the steps and the detail of the hierarchical structure give readability and meaning to the relationships, and commutativity to segments of the hierarchy so patterns and variations become clear.

When choosing whether to create a child step B to step A, as opposed to a peer step B to step A, for any two steps that begin (but, don't necessarily complete) in sequential order of first A and second B, consider one or both of these two questions:

1. Would the failure, blockage, or some other problem in step B cause a corresponding failure in step A? If yes, then B must be a child of A. If no, then step B follows step A as a sibling.
2. Does step B complete before step A is complete? If yes, then B must be a child of A. If no, then step B follows step A as a sibling.

At any node that is a parent node, completion of the step represented by the node might entirely consist of child nodes to the node, however declarations and/or definitions of automatic variables might not fit into child nodes due to scoping rules of the language.[14]

Higher-level, larger-scope steps always go towards the root (represented here as the top) of the hierarchy. Finer-grain steps with few or no dependencies naturally occur towards the leaf nodes of the hierarchy.

Rationale

For a complex set of steps that lay out a well-defined procedure, moving from a linear list of steps to an ordered-tree hierarchy brings these benefits:

1. Complex prose and even anything other than the simplest written grammar, with the subtleties and sometimes ambiguities inherent to the form, are no longer needed. Instead, the structure is explicit and unambiguous and contains only simple statements as elements of expression.
2. The relationships between the steps is expressed in a natural way, with peer steps and parent/child relationships, and the larger, more general scope steps close to the root node and the finer details (which people might choose to ignore) toward the leaf nodes.
3. The hierarchy assists in managing complexity around changes, failures or blockages.
4. Any amount of fine detail is available, or can be added as needed in sibling/peer or leaf nodes, with no accompanying changes to other parts of the hierarchy.
5. Any nodes in the hierarchy, and descendants and the descendant structures of that node, can be reused.

As Chuong Vo et al. wrote in their paper presenting a "TaskOS" for presenting tasks from the perspective of an actor and "in terms of their intend goals," [15]

> *Organising a task and its subtasks in a hierarchical structure is a natural solution for generating task-oriented interactions.*[16]

Closing

---

[14] Please see sample code available on the site http://MetaAutomation.net.
[15] Chuong C. Vo, Torab Torabi, & Seng W. Loke, "A Survey of Task Representations and Their Applicability to Smart Environments," (unpublished) http://homepage.cs.latrobe.edu.au/ccvo/papers/10SurveyOfTMDLs.pdf, page 23
[16] Ibid, page 7

For a presentation of ordered steps which supports a hierarchy, there are advantages in clarity, expressiveness and flexibility in presentation that result from using an ordered tree hierarchy rather than a linear list.

To the degree that any activity has context and steps, and relationship to or awareness of that context is a natural part of human activity, then the Hierarchical Steps pattern, too, is a natural part of human activity.