A View of the Restaurant Script Through the Lens of Hierarchical Planning

Jamie C Macbeth[1] JMACBETH@SMITH.EDU
Mark Roberts[2] MARK.C.ROBERTS20.CIV@US.NAVY.MIL
Boming Zhang[3] BOMINGZHANG@UMASS.EDU
Sharmin Badhan[4] SBADHAN213036@MSCSE.UIU.AC.BD
Molly Neu[1] AMALIA.R.NEU@GMAIL.COM
Tanush Garg[5] TANUSH_GARG@BROWN.EDU
Yining Hua[6] YININGHUA@G.HARVARD.EDU
Manushaqe Muco[7] MANJOLA@MIT.EDU
Mackie Zhou[8] MACKIEZHOU@MICROSOFT.COM

[1]Department of Computer Science, Smith College, 100 Green St., Northampton, MA 01063, USA
[2]Navy Center for Applied Research in Artificial intelligence, The U.S. Naval Research Laboratory, Washington, DC, 20375, USA
[3]University of Massachusetts, Amherst, MA, 01003, USA
[4]United International University, Madani Avenue, Badda, Dhaka 1212, 09604, Bangladesh
[5]Brown University, Providence, RI, 02912, USA
[6]Harvard University, Cambridge, MA, 02138, USA
[7]MIT, Cambridge, MA, 02139, USA
[8]Microsoft, Redmond, WA, 98052, USA

Abstract

The flexible nature of human cognition and of the structures it uses is well known, as is the difficulty of building cognitive systems that exhibit transfer and use the same structures for radically different tasks. In this paper, we perform a close examination of Schank-Abelsonian scripts, picking apart the goal- and plan- oriented nature of low-level acts and high-level reasoning inherent in them. We then view scripts through the lens of hierarchical planning systems and construct the well-known restaurant script as a hierarchical goal network planning domain. These are evidence in support of a claim that some, if not all, scripts are deeply hierarchical and are plan- and goal-oriented. The continuum that results from this representational unification may provide flexible knowledge structures which may be reused across a broad variety of tasks in language understanding, planning, and tasks requiring both, such as explanation and plan-based understanding of natural language.

1. Introduction

Despite the impressive engineering of recent machine learning models to address some forms of natural language communication, there remains a need to study flexible and reusable cognitive representations. The flexible nature of human cognition and of the structures it uses is well known, as is the difficulty of building cognitive systems that exhibit transfer and use the same structures for radically different tasks. The astounding discrepancy between the fluid, human-like output of large language models, the mainstream artificial intelligence systems at the time of this writing, and their poor performance on simple reasoning and planning problems (e.g., Valmeekam et al., 2024) could be attributed to representational issues.

Scripts are commonsense knowledge structures representing frequently-encountered situations which were originally used in story understanding systems (Schank & Abelson, 1977). Scripts are frequently revisited as a cognitive representation worthy of investigation, in part because they present an important alternative to logic and semantic networks for inference and reasoning. Although their noted limitations are that they are overly rigid and lack composability and transferability, there were proposals which suggested that these limitations could be overcome with goal and plan systems and processes. Unfortunately, these proposals were disconnected from work that evolved into the modern era research on automated planning systems.

Scripts are planning-like to those familiar with automated planning systems and their typical representations (e.g., PDDL (Haslum et al., 2019) is the de facto input language for deterministic planners, though there are others). However, scripts are inherently hierarchical and object-centric, emphasizing objects and roles. Recent advances in planning have standardized models for hierarchical planning (e.g., HDDL, Höller et al., 2020) but these languages have a strong assumption of a logic-centered modeling perspective rather than an object-centric perspective.

This paper addresses the above limitations by (1) examining the nature of the original script concept to uncover the kind of modeling that is needed and (2) encoding the restaurant script in a planning language that natively supports modeling in Python. For task (1), we perform a close examination of Schank-Abelsonian scripts, picking apart the goal- and plan- oriented nature of low-level acts and high-level reasoning inherent in the hierarchical structures of the scripts. We explore whether scripts have a hierarchical structure, and whether that hierarchy corresponds to goals and planning. For task (2), we leverage a planning modeling language to encode scripts that extends prior work in hierarchical goal networks (Shivashankar et al., 2012; Roberts et al., 2021).

We present our introspective findings and planning domain as evidence in support of a claim that some, if not all, scripts are deeply hierarchical and are plan- and goal-oriented. After providing some background (Section 2), the main contributions of this paper include:

- A description of an internal study where we examine the goal- and plan- structures of the restaurant script. The study investigates how the different features and structures of scripts may be combined with planning structures and systems to create combined flexible representations (Section 3).
- An encoding of the restaurant script in a planning language called OOMPA (Object-Oriented Modeling for Planning and Acting) and an introduction to OOMPA. This version of the restaurant script appears to account for the goal and planning knowledge that human reasoners have for restaurant situations (Section 4).

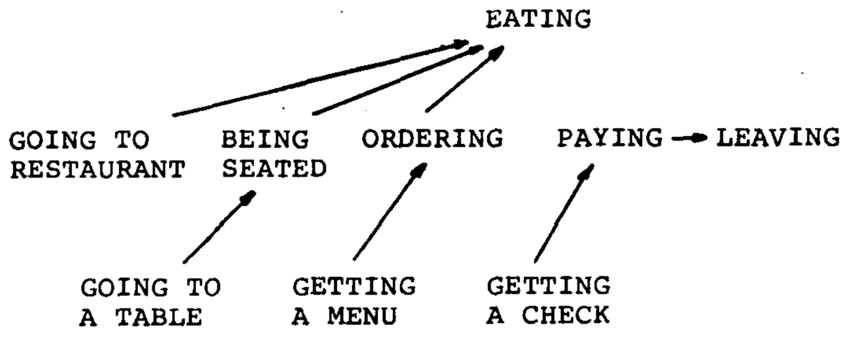

Figure 1. Lehnert static goal structure, from Lehnert (1975).

- A discussion of the challenges of aligning scripts with automated planning (Section 5).

Our contributions are initial steps towards a possible unification of script and plan representations to form a continuum of goal-oriented knowledge structures. This unification may provide more flexible knowledge structures which may be reused for a broad variety of tasks in language understanding, planning, and tasks requiring both, such as explanation and plan-based understanding of natural language. The work is also important in the ways that it relates automated planning tasks and systems to human episodic memory and explicit memory, and more broadly to human cognition. If one of the main critiques of scripts is that they are too rigid, then the flexibility of scripts and their components is achieved by leveraging the flexibility of the proposed planning model.

1.1 Running Example: The Restaurant Script (Schank & Abelson, 1977)

To describe the background and approach, we leverage the Restaurant script, which we will describe at an abstract level here. The description of scripts in Schank & Abelson (1977) divides the restaurant script into four *scenes*: Scene 1: Entering, in which the patron enters the restaurant, finds a place to sit, and sits; Scene 2: Ordering, in which the patron obtains a menu, chooses an item, and places an order with their server; Scene 3: Eating, in which the server delivers the item for the patron to eat; and Scene 4: Exiting, in which the server provides a check to the patron, and the patron pays, leaves a tip, and exits the restaurant. Each scene could have multiple sequences of *causal chains* of acts and events called *subscenes*, representing multiple ways of completing the activity.

A representation of the goals in this script, provided by Lehnert (1975), are shown in Figure 1. We examine this further in Section 3 after discussing some necessary background and related work.

2. Background

We outline two areas of prior research that we build on: Scripts and the Script Applier Mechanism (Section 2.1) and hierarchical planning (Section 2.2). Before we proceed, we clarify our usage tasks, activities, and goals. All three are equivalent because they can be converted to each other in polynomial time. In general, a goal is a boolean or numeric condition that evaluates to true when achieved.

For example, `patron.table` needs to non-nil to be achieved while `patron.money >= 50` compares the value of the patrons money against a concrete value. A task or an activity might achieve a goal, linked with a condition. For example, the `eating` activity could achieve `patron.is_sated`. The graph of Figure 1 specifies activities, despite being labeled a goal structure by Lehnert, and can be converted into goals: `going to restaurant` $\rightarrow$ `patron.location == restaurant`, `being seated` $\rightarrow$ `patron.table = table1`, and so on.

2.1 Scripts and SAM

Originally, scripts contained "causal chain" sequences of events, acts, and state changes that, in theory, reflect the human capability of "filling in the blank" during story understanding processes (Schank & Abelson, 1977). They contain much more structure than just the graph, but we will refer to the script as the composition of all the details. The first implementation of a script-based understander, the Script Applier Mechanism (SAM, Cullingford, 1977), was a symbolic AI system which applied scripts to the understanding of natural language narratives from a variety of sources.

In SAM, scripts are represented as sequences of events and acts in the Conceptual Dependency (CD) language-free meaning representation system (Schank, 1975) with *script variables* occupying *role* positions (e.g., ACTOR or OBJECT) in the script. SAM analyzes a text into CD form, matches these forms to the script, and binds the variables to people, objects, and places encountered in the story. SAM then instantiates a story representation consisting of the script structures, both those that matched to events in the story and those which did not, with variables replaced by their values, allowing SAM to instantiate implied events. SAM could compose summaries in English, Chinese, and Spanish and could answer questions about the texts as a way to demonstrate its context-dependent common-sense knowledge of social interactions and human affairs (Lehnert, 1975).

The scenes and events described by Schank & Abelson (1977) are for a particular *track* in the restaurant script that represents a style of restaurant dining called the "Coffee Shop Track". Other script tracks corresponded to manifestations of the social situation which differed slightly (for example, fast-food restaurant dining would occupy a separate track of the script).

Scenes in SAM had a *main conceptualization* (maincon), which was the most important or crucial act or event of the scene. Separately, each script had a script-level maincon. The script applier used maincons to generate its own summaries of stories that it understood by generating natural language from the maincons of each script scene (Cullingford, 1977). In Section 3, we will examine how maincons play a role in setting the expectations, and thus, the goals, of the script.

SAM could also answer some WHY questions which explained the goal-oriented behaviors of story actors. To answer these kinds of questions, SAM used a hierarchical static goal graph as shown in Figure 1. The graph consists of scriptgoals and a set of subgoals. The highest-level goal of the restaurant script was eating, while being seated and ordering were subgoals of eating. A question about the goal of an action at a particular node was provided by traversing the edge to the next higher node in the hierarchy. For example, if the question WHY DID MARY GET A MENU is asked, SAM followed the edge to "ordering" to provide the answer '*because she was ordering*'. This worked for questions regarding the actions of the restaurant customer, but was brittle in the face of "weird", non-script based unexpected occurrences in the story (Lehnert, 1975).

Schank & Abelson (1977) wrote “there is a fine line between the point where scripts leave off and plans begin. In a sense it is an unimportant distinction” (page 77). Schank & Abelson proposed separate structures to represent goals and plans (Schank & Abelson, 1977) which were implemented in the Plan Applier Mechanism (Wilensky, 1983). However, PAM did not integrate the capabilities of SAM with respect to understanding highly stereotypical situations handled well by scripts. Later work acknowledged “the line between scripts and plans seemed fuzzy” and focused on abstractions for generalization and learning, building so-called “sketchy scripts” which admittedly “contain less information” (Schank, 1982). This paper seeks to overcome some of these limitations by combining plan-oriented representations with script-oriented representations, allowing a continuum of abstractions with reusable knowledge.

2.2 Hierarchical Planning

Most automated planning systems perform forward simulation of actions to search for a reachable goal condition. In automated planning, the world (called a *domain*) is represented as an implicit state transition system (Ghallab et al., 2025). The domain describes object types with their attributes and relationships, which are often modeled using predicates–following from first-order logic. Instead, we represent them as *state-variables*, which are more programming like. For example, suppose the patron `pat` of a restaurant is assigned the server `sam`. PDDL represents this as `(server pat sam)` read as the `server` of `pat` is `sam`. As a state variable, it is `pat.server == sam`. The two representations are expressively equivalent and can be converted to each other in polynomial time (Ghallab et al., 2025), but the state-variable representation is much easier for most programmers.

Actions are functions that modify attributes or relations. For example, sitting at a table would result in the patron being assigned a table and its server as well as the table being occupied. Together these attributes, relations, and actions describe the implicit state transition system. The transition system is often too large to fit in memory, so techniques such as search are used to find a solution.

A *hierarchical planning system* (Nau et al., 1999) adds domain knowledge, in the form of recipes, to improve search efficiency. Hierarchical planners are provably more expressive than classical planning (Erol et al., 1994). In contrast to classical planning systems, which construct a plan from the initial state to the goal state through simple state-space search, hierarchical planning systems solve the problem by recursively breaking it down into intermediate steps to determine what actions an intelligent agent, robot, or other autonomous systems should take to meet its goals in the real world. Decomposition is done through the use of *methods*. A hierarchical problem description consists of the domain, including methods, the initial state, and the desired goal condition.

Hierarchical Task Network (HTN) planning systems such as the Simple Hierarchical Ordered Planner (SHOP) planning system (Erol et al., 1994; Nau et al., 1999, 2003) are often used for applications where fast planning is desired and domain-dependent knowledge can make the planning process more efficient. A newer variant of hierarchical planning, called Hierarchical Goal Networks (HGNs, Shivashankar et al., 2012, 2013) decomposes *goals*. Formally, HTNs and HGNs are equivalent (Alford et al., 2016). Practically, HGNs are easier to integrate with classical planning heuristics and planners, which is important if the methods are incomplete.

3. Goal-Oriented Thinking in Scripts

To understand how to model scripts with planning, we present introspective evidence for goals linked to the restaurant script. We performed an analysis of our structured discussions about activities in restaurants and the plan-like behaviors of those activities. The analysis informs the construction of the hierarchical goal network planning domain discussed in Section 4. In general, we were able to associate each act in the restaurant script with a goal of some kind, even for concrete acts. In searching for patterns, we uncovered elements of a "bottom-up" construction of a goal network for the script, and found a strong link between subgoals and maincons.

Methodology. To examine the plans and goals associated with scripts, we discussed the goal orientation of acts within script structures. We chose the restaurant script because it is well known and has a full CD description, divided into four scenes (Schank & Abelson, 1977).

Table 1 shows our English realizations of the 33 CD acts in Schank & Abelson's (1977) detailed description of the restaurant script. This conversion alleviated the need for familiarity with CD structures. In creating the English realizations, we had to include descriptions of the roles for clarity. For example, the "F-Food" role was "the food item that the customer ordered."

We constructed a questionnaire with two types of questions: activity goal or scene maincon. 33 questions ask about the ultimate goals and purposes of each act. We constructed these by turning the English realizations into WHY questions. For example, THE CUSTOMER LOOKS AT THE TABLES was transformed to WHY DOES THE CUSTOMER LOOK AT THE TABLES? Four questions explored the scene maincon. These questions listed all acts in a scene and asked, for each scene, WHICH IS THE PRINCIPAL OR MOST IMPORTANT ACT? We allowed multiple answers, and for shorter scenes, particularly "Scene 3: Eating", which only contains three acts, we included a few acts from neighboring scenes in the list of acts to choose from. We randomized the order of the questions to minimize ordering effects.

We individually answered the questionnaire; each person had varying levels of familiarity with scripts and with the restaurant script in particular. We shared our questionnaire answers with each other and discussed them, analyzing our responses to the questionnaire and using them to make findings. The sections that follow include quotations from our answers and discussions. QUESTIONS ARE IN THIS STYLE. '*Answers are in this style.*'

3.1 Goals, Subgoals, and Scriptgoals

First, we generally found that we were all able to construct answers to these questions, although the WHY questions that we asked ourselves in the questionnaire were, at times, awkward because they attempted to access goal-oriented knowledge, which is rarely expressed so directly. We were able to associate each act in the restaurant script with a goal of some kind, even for very concrete acts. For example, when we asked ourselves WHY DOES THE CUSTOMER GO TO A TABLE? most of us said '*to sit at it*' or mentioned sitting. This would align with Schank's "redefinition" of scripts (Schank & Abelson, 1977) in his later writing on *Dynamic Memory* (Schank, 1982) where he writes, "A scene defines a setting, an instrumental goal, and actions that take place in that setting in service of that goal."

We hypothesized that if our restaurant script knowledge was a kind of hierarchical network, concrete subgoals might appear in service of more abstract goals. We first thought that the goals of acts might conform to Lehnert's hierarchical graph (Figure 1). In this model, the goals of acts within

Table 1. Scenes of the restaurant script (Schank & Abelson, 1977), the Conceptual Dependency (CD) representations of the events that characterize the scene, and English descriptions of the events. Duplicates of events which appeared more than once in different scenes (e.g. "The waiter goes to the customer's table") have been left out, along with path information.

Scene	CD	English
Entering	S PTRANS S into restaurant	The customer goes into a restaurant
	S ATTEND eyes to tables	The customer looks at the tables
	S MBUILD where to sit	The customer decides where to sit down.
	S PTRANS S to table	The customer goes to a table.
	S MOVE S to sitting position	The customer sits down at a table.
Ordering	S PTRANS menu to S	The customer picks up the menu
	S MTRANS 'need menu' to W	The customer tells the waiter that they need a menu.
	W PTRANS W to menu	The waiter goes over to where the menus are.
	W PTRANS W to table	The waiter goes to the customer's table
	W ATRANS menu to S	The waiter gives a menu to the customer
	S MTRANS food list to CP(S)	The customer reads the menu
	S MBUILD choice of F	The customer chooses a food item.
	S MTRANS signal to W	The customer signals the waiter.
	S MTRANS 'I want F' to W	The customer tells waiter the food item that they want.
	W PTRANS W to C	The waiter goes to the cook.
	W MTRANS (ATRANS F) to C	The waiter tells the cook to give them the food item that the customer ordered.
	C MTRANS 'no F' to W	The cook tells the waiter that F is not available
	W MTRANS 'no F' to S	The waiter tells the customer F is not available
	C DO (prepare F script)	The cook prepares the food item that the customer ordered.
Eating	C ATRANS F to W	The cook gives the food item that the customer ordered to the waiter.
	W ATRANS F to S	The waiter gives the food item that the customer ordered to the customer.
	S INGEST F	The customer eats the food item that they ordered.
Exiting	S MTRANS to W (W ATRANS check to S)	The customer asks the waiter for the check.
	W MOVE (write check)	The waiter writes or prints a check for the customer.
	W ATRANS check to S	The waiter gives the customer the check.
	S ATRANS tip to W	The customer gives the waiter a tip.
	S PTRANS S to M	The customer goes to the cashier.
	S ATRANS money to M	The customer gives money to the cashier.
	S PTRANS S to out of restaurant	The customer leaves the restaurant.

scenes (Table 1) would be the scene subgoals—being seated, ordering, and paying. We collectively thought that this was frequently the case.

According to this model, the goal of a scene subgoal act would be a more abstract goal, which Lehnert termed a *scriptgoal*. Figure 1 has "eating" as the scriptgoal of the scene subgoal acts such as "going to restaurant," "being seated," or "ordering." When asked WHY DOES THE CUSTOMER GO INTO THE RESTAURANT, we said '*to eat*' or '*to eat a meal,*' referencing an abstract scriptgoal. However, occasionally a different act was elevated to a scriptgoal. For ordering, WHY DOES THE CUSTOMER TELL THE SERVER THE FOOD ITEM THAT THEY WANT? focused on having the item '*prepared,*' '*delivered,*' or '*brought*' by the server; only one of us focused on eating, the presumed answer based on Lehnert's diagram.

Further evidence of a goal hierarchy were answers to questions with multiple goals in the hierarchy. Our answers mentioned both the scene subgoals and the abstract goals. For example, in the Entering scene our answers to WHY DOES THE CUSTOMER GO TO A TABLE? mentioned sitting and the purpose of eating: '*to sit down and eat*' or '*To order and eat at the table.*' In the Exiting scene, for WHY DOES

THE CUSTOMER ASK THE SERVER FOR THE CHECK? or WHY DOES THE CUSTOMER GO TO THE CASHIER? many answers mentioned multiple goals: '*the customer wants to pay for their meal and leave.*'

Sometimes subgoals other than the expected scene subgoals appeared. In the Entering scene, for WHY DOES THE CUSTOMER LOOK AROUND AT THE TABLES IN THE RESTAURANT? some answered '*being seated*' as might be expected. But we also stated '*to select,*' '*to decide,*' '*to pick,*' and to '*find a good place to sit,*' referring to the act of choosing where to sit before sitting. Similarly, in the ordering scene, to WHY DOES THE CUSTOMER TELL THE SERVER THAT THEY NEED A MENU?, WHY DOES THE CUSTOMER PICK UP THE MENU?, or WHY DOES THE CUSTOMER READ THE MENU? answers included '*to decide what to order,*' '*to select,*' '*to figure out what they want,*' etc. Thus, there may be more levels to the hierarchy than originally expected.

In the Eating scene, the only act by the customer is eating. We largely discussed satisfying hunger and appetite as the goals related to the eating act, along with the positive taste sensation being a goal: '*they think the food is tasty so they want to eat it.*' We also talked about the original intention or reason for going to the restaurant: '*they ... came to the restaurant with the intention of eating the food that they ordered,*' '*because they came to the restaurant to eat,*' '*to fulfill the reason they went to the restaurant for.*' This confirms that eating is the most abstract goal, or scriptgoal, or at least it is *an* abstract goal for the customer.

3.2 Server and Cook Acts

Of the 33 acts in the restaurant script diagram provided by Schank & Abelson (1977), 17 of the acts are by the customer, but 13 of the acts are by the waiter, and three are by the cook. The goals of server acts often center on the server and cook assisting the customer. For WHY DOES THE SERVER GIVE THE FOOD ITEM THAT THE CUSTOMER ORDERED TO THE CUSTOMER? many of us talked about the customer's ultimate goal of wanting to eat or enjoy the food: '*The customer can enjoy the meal*', '*so the customer can eat it*'. But we also discussed the server's obligation to perform this act as part of their job: '*That's part of what they are paid to do*'.

But in other cases, it was clear that the server had their own hierarchical goals and that the goals of their acts were abstract subgoals for other server acts. For WHY DOES THE SERVER GO OVER TO WHERE THE MENUS ARE? we mainly thought it was '*to get a menu to give to the customer*'. When we asked WHY DOES THE SERVER GO TO THE COOK? among other answers, we said, '*to pick up food*'.

3.3 Are Maincons Goals?

As a reminder, a maincon is the main conceptualization—the most important or crucial act or event of a scene or of a script. Although the names of scenes, "Entering," "Ordering," "Eating," and "Exiting," lead to natural hypotheses about their maincons, we wanted to make our own determinations. The questionnaire asked about the most important act in the Ordering scene, to which the majority said '*The customer tells the server the food item that they want*'. For many of us, the customer eating the food item that they ordered was a maincon of "Eating". One of us wrote: '*To me this is the most important because it is the primary reason the customer entered the restaurant*'. For the Exiting scene, we overwhelmingly chose "the customer gives money to the cashier" as the most important act, with one of us stating '*this is the most important ... because the customer has eaten a meal and must pay before leaving*'. We found that generally, the maincons of scenes were identical to the subgoals of scenes.

However, there were exceptions to this that prompted us to reimagine the structure of the script. In our discussions of "Entering", we found "the customer goes into a restaurant" to be a more important act than "the customer sits down at a table". Entering the restaurant was seen as most important '*because the customer was probably hungry and this is the first commitment they made to eating in a restaurant. They could have eaten food in other ways (e.g., getting food at a grocery store or from their home).*' For Eating, many believed that the cook preparing the food item, or the server giving the food item to the customer were crucial or even more important than the act of eating. One of us wrote '*food being delivered ... was the main point of the visit to the restaurant.*' This is a case where the most abstract maincon mismatched what we discussed as most abstract goal of the script.

We discussed the way that the "transaction" between the restaurant and the patron, in which the patron pays and the restaurant prepares and delivers food, was more important than the act of eating, particularly when seen from the server's perspective. We also talked about "tracks" of the restaurant script for take-out restaurants and the possibility that the patron might not actually consume some or all of the food. Furthermore, having "completing the transaction" as the high-level goal also allows for the payment that occurs in the "Exiting" scene to be in service of that goal. We also noticed that, if we redefined the script scene structure taking into consideration Lehnert's graph (Figure 1), then the act of entering the restaurant could be in a separate "Entering" scene, and the goal of sitting down would correspond to the maincon of a new "Getting Seated" scene.

4. A Script-Based Planning Domain

To illustrate the possibility of a joint script and planning representation, we represented the restaurant script as a planning domain. The situation is very much unlike that in the script applier mechanism, which represents the world only in the form of a causal chain of events, and the existence of certain actors, objects, and settings as script variables.

4.1 The OOMPA Modeling language

We construct the domain as a hierarchical goal network[1] and developed in OOMPA (Object-Oriented Modeling for Planning and Acting), which is a toolkit to support modeling planning domains in a Pythonic and object-oriented manner. OOMPA blends the principles of ActorSim (Roberts et al., 2021; Johnson et al., 2016) with the ease and simplicity of the GTPyhop[2]. OOMPA's design makes writing Python planning models much more straightforward by: (1) leveraging the typing and introspection modules from Python 3.13, the latest stable release of Python; (2) supporting object-oriented modeling patterns in contrast to the usual declarative logic representations typical in automated planning languages; and (3) providing Python annotations (similar to the Python property) that allow programmers to annotate objects to link them to a planning system. Together, these design choices should enable anyone who knows Python to write planning models.

Below, we relate OOMPA representation with elements of the restaurant script, specifically the Patron; a full listing is provided in Appendix A. OOMPA organizes information using standard Python classes annotated with StateProperties, Actions, and Methods, described next. We changed

1. We provide examples in this section and Appendix A, but will release the full code when published.
2. https://github.com/dananau/GTPyhop

```
1  class Patron(Person, CreatesNewObjects, HasOompaActions, HasOompaMethods):
2    table: Table | None = StatePropertyFactory(None)
3    server: Server | None = StatePropertyFactory(None)
4
5    @OompaAction  # declares action ----------------------------------------
6    def sit(self, table: Table): pass
7
8    @sit.precondition  # condition for this action to be applicable
9    def sit(self, table: Table): return self.table.equals(None)
10
11   @sit.effect  # change in state from applying this action to state
12   def sit(self, table: Table):
13     return AndEffect( self.table.assigned(table),  # Patron seated at table
14                       self.server.assigned(table.server),  # server assigned
15                       table.occupied.assigned(True))  # table occupied
16
17   @OompaMethod  # declares method ----------------------------------------
18   def m_ordering(self, table: Table) -> GoalMethod: pass
19
20   @m_ordering.head  # the condition the method matches; determines relevance
21   def m_ordering(self, table: Table) -> Condition:
22     return AndCondition( self.desired_order.not_equals(None),
23                          self.desired_order.status.equals(ORDERED))
24
25   @m_ordering.precondition  # condition for this method to be applicable
26   def m_ordering(self, table: Table) -> Condition:
27     return AndCondition( self.table.not_equals(None),  # Patron seated
28       table.menu.not_equals(None),  # Patron lacks menu
29       self.desired_order.equals(None),) # Patron lacks desired order
30
31   @m_ordering.body  # change in hgn from applying this method to hgn
32   def m_ordering(self, table: Table) -> TOHGN:
33     return TOHGN( self.pickup_menu(), self.review_menu_for_special(),
34                   self.request_server(), self.place_order() )
```

Figure 2. A condensed version of the Patron class with the sit action and the order special method.

the names of roles from the original restaurant script to use phonetically matched starting consonants. Instead of S as Customer, Pat is the Patron; instead of W as Waiter, Sam is the Server; instead of C as Cook, Chris is the Cook.

Objects in OOMPA are standard Python classes, as shown in the condensed version of the Patron class in Figure 2 (for the full domain, see Section A). The Patron class (Line 1) inherits from the Person class, which is only a type, as well as some OOMPA declarations that this class creates new objects, has actions, and has methods. StateProperties are managed attributes that are declared using a `StatePropertyFactory`. The Patron lists two StateProperties (Lines 2-3) of the table and the server. A `StateProperty` works similarly to the Python property descriptor and allows OOMPA to infer the value type as well as automatically construct the state.

State in OOMPA is represented as a dictionary of dictionaries, as shown in Figure 3. Objects like this are converted to the dictionary state by unpacking the attribute names for each object and entering the value of the attribute. So a state is indexed by the attributes objects can take (e.g.,

```
s_0:
  check: {'sam': None}
  cost: {'omelette': 12}
  desired_order: {'pat': None}
  is_hungry: {'pat': True}
  is_pleased: {'pat': False}
  items: {'menu_breakfast': []}
  menu: {'pat': None, 'sam': None,
         'table1': menu_breakfast}
  money: {'pat': 50}
  near_to: {'sam': None}
  occupied: {'table1': False}
  order: {'sam': None}
  ordered: {'chris': None}
  prepared: {'chris': []}
  server: {'pat': None, 'table1': sam}
  special: {'menu_breakfast': omelette}
  status: {'omelette': AVAILABLE}
  table: {'pat': None}
```

Figure 3. The initial state s_0 represented as a dictionary.

```
  s_final: desired_order: {'pat': ordered_omelette}
           menu: {'pat': menu_breakfast}
           near_to: {'sam': pat      }
           occupied: {'table1': True}
           order: {'sam': ordered_omelette}
           server: {'pat': sam, 'table1': sam}
           status: {'ordered_omelette': 'ORDERED'}
           table: {'pat': table1}
```

Figure 4. The final state difference after applying (`m_sit`,`m_ordering`)

the table or server of the patron, the order being processed, whether the patron is pleased) and the names of the objects. For example, to determine whether patron Pat is hungry in state s_0 one retrieves s_0`[is_hungry][pat]`.

Actions and Methods are declared using annotations (i.e., @OompaAction and @OompaMethod). An action includes the parameters of the action, a precondition, and an effect. A patron can `sit` (Lines 6-15) at the table if the table is currently assigned `None`. The effect of `sit` assigns `table` to `patron.table`, `table.server` to `patron.server`, and `True` to `table.occupied`.

The ordering scene is represented in a method `m_ordering` (Lines 20-34), which matches the condition for `patron.desired_order` to have a value with status `ORDERED`. Its precondition is that these are not yet set; applying this method decomposes the goal into the sequence of actions `(pickup_menu, review_menu_for_special, request_server, place_order)`.

Solutions. A solution to problem is a plan, a sequence of actions that starts in the initial state s_0 and achieves the goal condition, similar to how SAM applies a script to a story. An automated planner constructs a solution by applying methods and actions. OOMPA implements a variation of the Goal Decomposition Planner (Shivashankar et al., 2012).

In the restaurant domain, `StateProperties` represent the locations of roles and objects. The `StateProperty` for `is_hungry` and `is_pleased` (not shown Figure 2 but listed in Section A) are part of the entry conditions and results of the original restaurant script (Schank & Abelson, 1977). Other entry and result conditions related to money are represented by `cost`—

the cost of menu items—and `money`, which indicates how much money the patron has. The `near_to` property indicates the proximity of things. The location or possession of the menu is designated `menu`; it is initially on the table. The customer orders the *special* menu item of the day, the `veggie_omelette`. Let the `StateProperty` x = `desired_order(pat)`. For the goal ($x! =$ `None` $\wedge$ `status(x) == ORDERED`), GDP uses `m-sit` and `m-ordering` and returns the solution: `sit(pat, table1)`, `pickup_menu(pat)`, `review_menu_for_special(pat)`, `request_server(pat)`, and `place_order(pat)`.

5. Discussion

Early conceptions of scripts focus on defining them as structures of episodic memory of personal experience consisting of causal chain sequences of acts and events which occurred at a particular time and place (Schank & Abelson, 1977). Scripts pose structures based on personal experiences as an alternative to logical formulations as a basis for thinking and reasoning. Schank & Abelson (1977) counterpose scripts and episodic memory as "flat" sequential structures against semantic memory, which represents general knowledge facts often organized around words and their meanings using hierarchies of ontological class membership and property inheritance.

Our introspective discussions and hierarchical planning domain are evidence of the hierarchical nature of scripts that exists alongside the sequential order implied by episodic memory. In fact, we found (in Section 3) that the subgoals of scenes might extend to greater depths than previously envisioned (e.g., choosing a place to sit as a subgoal of sitting down, or choosing a food item from the menu as a subgoal to making an order). This view aligns with elements of the Script Applier Mechanism implementations (Lehnert, 1975; Cullingford, 1977) and with later proposals by Schank (1982) on abstraction and generalization in relation to scripts.

There is certainly goal-oriented thinking in the individual acts of scripts, but what kind of planning structure is a script as a whole? In *Scripts, Plans, Goals, and Understanding*, Schank & Abelson (1977) wrote "the restaurant ... script is known to be a common means of implementing a plan of action for getting fed" (page 49), "a routinized plan can become a script" (page 72), and "plans are where scripts come from" (page 72). The key difference between script application and planning would appear to be that scripts and script application can be successful in domains where there is a strong level of expectation about what will happen, and little variation to the effectiveness of stereotyped sequences of actions. If we consider a entirety of a script which contains its scene, subscene, and track structures, and attempt to relate it to hierarchical planning, a script appears to be most similar to a *decomposition tree* (also termed *solution tree*), which is the tree-like structure created by the planner that includes subgoals and choices of methods at each stage.

When drawing further correspondences between the restaurant script and a hierarchical goal network, we see that, although the scenes of the restaurant script as described by Schank & Abelson (1977) are not explicitly described as a decomposition of "having eaten in a restaurant", as a first pass, our planning domain does decompose the task into subgoals corresponding to the restaurant scenes. The methods `m_entering`, `m_ordering`, `m_eating`, and `m_exiting` would roughly corresponding to the four scenes of the restaurant script. Also, for Cullingford (1977), script scenes were allowed to have multiple "subscenes", sequences of events and acts representing possible ways of accomplishing the activity. HGN methods are analogous to script subscenes in this light as well, in that there may be multiple methods which are applicable in a particular state, with the bodies of the methods containing low level actions for reaching a subgoal.

In a way very different from script scenes, however, each method expresses the goal state that is meant to be reached by it, along with preconditions, the aspects of the state that must hold in order for the method to be used. This restriction, typical of hierarchical planning, allows the planner to select only methods that are applicable to the current state of the world; it allows methods to be reused in similar situations.

In some similarity to methods in HGNs, the original description of the restaurant script (Schank & Abelson, 1977) had pre- and post- conditions (named "Entry conditions" and "Results"). Additionally, scripts in the Script Applier Mechanism were said to have a set of conceptualization patterns called "headers" which "invoked" the script and applied it to a story (Cullingford, 1977). However, in these early incarnations in the literature script structures had no mechanisms for establishing that the state of the world was appropriate for a particular scene to begin. Further, the ordering of scenes was fixed, and there were no built-in mechanisms for the reuse of scenes. Later Schank (1982) proposed memory organization packets (MOPs), high-level structures which allow scenes to be reused and shared across different settings. Methods within a hierarchical planner could be viewed as a concrete form of MOP implementation.

The Script Applier Mechanism used scripts for story understanding, but also for question answering, summarization, and translation. Although natural language systems for interacting with hierarchical planners exist, deepening the connections between scripts and planning domains as we have demonstrated could improve these capabilities. Specifically, the low-level causal chains in combination with maincons at the script and scene levels enable the generation of complex natural language explanations of plans, of the planning domain's structures, or of aspects of the planning process, such as its success or failure.

6. Related Work

Our motivation is exploring hierarchical planning and story understanding, for which Scripts and CD are central. Scripts–a kind of plan template–are widely applicable to problems resembling plan generation or plan understanding, and a variety of research has applied automated planning techniques to problems in these areas. We briefly mention some connections here and welcome further pointers.

Cognitive Systems. Scripts and methods both encode what would be called procedural knowledge in a cognitive system. Two systems bear the strongest overlap with the OOMPA encoding. SOAR (Laird, 2012) is widely known for its universal subgoaling, a decomposition technique similar to methods in OOMPA. Kirk & Laird (2019) used SOAR to learn games and the resulting models share similarities to OOMPA. The PUG (Planning with Utilities and Goals) system of Langley et al. (2016; 2024) has a similar focus on the primacy of goals for driving system behavior as OOMPA and its predecessor ActorSim (Roberts et al., 2021). Macbeth & Roberts (2018) previously demonstrated a connection between hierarchical planning and primitive decomposition.

PAIR. Story understanding bears resemblance to the literature on PAIR (plan, activity, intent, recognition), since understanding requires matching a trace with possible plans (or scripts). Geib et al. uses grammars to perform recognition (2011; 2015; 2018). Other approaches often adopt the compilation paradigm by Ramirez & Geffner (2009) of recognition as planning. Similarly related is the work that combines PAIR with analogy (e.g., Rabkina et al. (2022)).

Narrative. Riedl & Young (2010) used a plan-space planner–it searches the space of plans as opposed to states–to generate narrative plots. A compilation approach by Haslum (2012) showed how story generation can be compiled to a classical planning problem. An excellent survey of the area by Cardona-Rivera et al. (2024) even advocates for the use of hierarchical planning to capture typical procedures while increasing authorial control of temporally extended goals.

Dialog. Finally, researchers have studied how to use planning to generate dialog, often in the context of social agents or social robotics. The closest work to ours is by Cavazza et al. (2002), which uses an HTN planner to generate dynamic stories in the face of interruption. Porteus et al. (2009; 2011) extended this work to include a user-interface and the ability to modify constraints. Petrick & Foster (2013) built conversational social robots, where knowledge structures representing human goals and plans make robotic interactions and the character actions in texts and interactive games more believable and easier to understand.

7. Conclusion and Future Work

In this paper we carefully examined Schank & Abelson's restaurant script, and used what we learned about its hierarchical structures and the goal- and plan- oriented nature of its low-level acts to encode it in a hierarchical goal network planning domain. We demonstrated how reasoning that had previously been associated with story understanding using the restaurant script could be posed as a planning problem in our domain. This is an initial step towards unification of scripts with the knowledge structures of hierarchical planning, which reflects the flexible structures that humans reuse in performing the radically different tasks of planning and story understanding. Future work should validate our discussion findings with a full human-subjects study of the relationship between goals, tasks, and activities in various scripts. Furthermore, the OOMPA implementation could be more general; future work should attempt to represent these problems using standard planning languages like PDDL and extend what both representations are encoding.

Two other major issues interest us for future work in this area, which will be bolstered by efforts to build a replica of the Script Applier Mechanism combined with a hierarchical planner. Firstly, the original scripts contained mechanisms for dealing with what were termed "goal interference" situations in which there are minor, but somewhat expected, obstacles to the smoothest flow of events. For example, arriving at a restaurant and observing that there are no available tables was considered an interference with the goal of being seated (Lehnert, 1975). Implementations described in the literature (Cullingford, 1977; Lehnert, 1975), had scripts which contained multiple alternative paths which contain occurrences of goal interference and predicted an subsequent event which would be either a resolution or consequence of the interference. For example, the restaurant customer may eventually be seated at a table after a short wait.

These aspects of scripts appear to relate strongly to certain constructs and concepts in the literature on automated planning and acting. Prior work recognizes the many ways in which the expectations of a plan can diverge from reality while executing a plan, creating situations in which the original plan must be adapted to a new situation (Fox et al., 2006). Further research is needed on the connections between script interferences and concepts of *plan repair*, in which an existing plan is adapted to a new context, and *replanning*, in which a new plan is generated completely from the ground up. If we view a script as a plan, an interference in a script can be viewed as a replanning or plan repair situation.

Secondly, we see characteristics of scripts which may suggest novel features for automated planning systems. As we mentioned, scripts in the Script Applier Mechanism were supplied with multiple paths in scenes that represented different ways that the main point or main goal of the scene is accomplished. They also contained interference paths which were expectations for certain kinds of things that could "go wrong" in a scene, and expected ways that the obstacle could be overcome. The fact that these structures exist for scripts suggests a possible improvement to planning systems in which they construct plans with built-in expectations of the ways in which the plan could be thwarted, along with pre-built replans and plan repairs for the initial plan which handle these expected obstacles; In the planning literature these are typically called *plan repair* (e.g., Fox et al. (2006); Zaidins et al. (to appear)) and *contingent planning* (e.g., Hoffmann & Brafman (2005)). Future work can explore this aspect of the representational unification of scripts and hierarchical planning.

Acknowledgements

MR thanks NRL for funding this research.

References

Alford, R., Shivashankar, V., Roberts, M., Frank, J., & Aha, D. W. (2016). Hierarchical planning: relating task and goal decomposition with task sharing. *Proc. of IJCAI*. New York, New York, USA: AAAI Press.

Cardona-Rivera, R. E., Jhala, A., Porteous, J., & Young, R. M. (2024). The story so far on narrative planning. *Proc. of ICAPS*, *34*, 489–499.

Cavazza, M., Charles, F., & Mead, S. (2002). Character-based interactive storytelling. *IEEE Intelligent Systems* (p. 17–24).

Cullingford, R. E. (1977). *Script application: Computer understanding of newspaper stories*. Doctoral dissertation, Yale University, New Haven, CT.

Erol, K., Hendler, J. A., & Nau, D. S. (1994). UMCP: a sound and complete procedure for hierarchical task-network planning. *Proc. of AIPS* (pp. 249–254). Chicago, IL: AAAI Press.

Fox, M., Gerevini, A., Long, D., & Serina, I. (2006). Plan stability: Replanning versus plan repair. *Proc. of ICAPS* (pp. 212–221). AAAI.

Geib, C. (2015). Lexicalized reasoning. *Proc. ACS*. Atlanta, GA: The Cognitive Sys. Foundation.

Geib, C., & Goldman, R. (2011). Recognizing plans with loops represented in a lexicalized grammar. *Proc. of AAAI*, *25*, 958–963.

Geib, C. W., & Kantharaju, P. (2018). Learning combinatory categorial grammars for plan recognition. *Proc. of AAAI* (p. 3007–3014). New Orleans, Louisiana, USA: AAAI Press.

Ghallab, M., Nau, D., & Traverso, P. (2025). *Acting, planning, and learning*. Cambridge University Press. Authors preprint at https://projects.laas.fr/planning/.

Haslum, P. (2012). Narrative planning: Compilations to classical planning. *JAIR*, *44*, 383–395.

Haslum, P., Lipovetzky, N., Magazzeni, D., & Muise, C. (2019). *An introduction to the planning domain definition language*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Cham: Springer International Publishing.

Hoffmann, J., & Brafman, R. I. (2005). Contingent planning via heuristic forward search with implicit belief states. *Proc. of ICAPS*.

Höller, D., Behnke, G., Bercher, P., Biundo, S., Fiorino, H., Pellier, D., & Alford, R. (2020). Hddl: An extension to pddl for expressing hierarchical planning problems. *Proc. of AAAI*, *34*, 9883–9891.

Johnson, B., Roberts, M., Apker, T., & Aha, D. W. (2016). Goal reasoning with information measures. *Proc. of ACS*. Evanstan, IL: Advances in Cognitive Systems.

Kirk, J. R., & Laird, J. E. (2019). Learning hierarchical symbolic representations to support interactive task learning and knowledge transfer. *Proc. of IJCAI* (p. 6095–6102). Macao, China.

Laird, J. (2012). *The soar cognitive architecture*. Cambridge, MA: MIT Press.

Langley, P., Barley, M., Meadows, B., Choi, D., & Katz, E. P. (2016). Goals, utilities, and mental simulation in continuous planning. *Proc. of ACS*.

Langley, P., Barley, M., Meadows, B., Choi, D., & Katz, E. P. (2024). A unified cognitive architecture for embodied intelligent agents. *Proc. of ACS*.

Lehnert, W. (1975). What makes SAM run? script based techniques for question answering. *Proc. Workshop on Theoretical Issues in NLP* (p. 16–21). USA: Assoc. for Comp. Ling.

Macbeth, J. C., & Roberts, M. (2018). Exploring connections between primitive decomposition of natural language and hierarchical planning. *Proc. of ACS*. Stanford, CA: Cog. Sys. Foundation.

Nau, D., Cao, Y., Lotem, A., & Muñoz-Avila, H. (1999). SHOP: Simple hierarchical ordered planner. *Proc. of IJCAI* (pp. 968–973). Stockholm: Morgan Kaufmann Publishers, Inc.

Nau, D. S., Au, T.-C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., & Yaman, F. (2003). SHOP2: an HTN planning system. *JAIR*, *20*, 379–404.

Petrick, R., & Foster, M. E. (2013). Planning for social interaction in a robot bartender domain. *Proc. of ICAPS*. Rome: AAAI Press.

Porteous, J., & Cavazza, M. (2009). Controlling narrative generation with planning trajectories: The role of constraints. *Interactive Storytelling* (p. 234–245). Berlin, Heidelberg: Springer.

Porteous, J., Teutenberg, J., Pizzi, D., & Cavazza, M. (2011). Visual programming of plan dynamics using constraints and landmarks. *Proc. of ICAPS* (p. 186–193).

Rabkina, I., Kantharaju, P., Wilson, J. R., Roberts, M., & Hiatt, L. M. (2022). Evaluation of goal recognition systems on unreliable data and uninspectable agents. *Frontiers in AI*, *4*.

Ramirez, M., & Geffner, H. (2009). Plan recognition as planning. *Proc. of IJCAI*.

Riedl, M. O., & Young, R. M. (2010). Narrative planning: Balancing plot and character. *JAIR*, *39*, 217–268.

Roberts, M., Hiatt, L. M., Shetty, V., Brumback, B., Enochs, B., & Jampathom, P. (2021). Goal lifecycle networks for robotics. *Proc. of FLAIRS*.

Schank, R. C. (1975). *Conceptual information processing*. New York, NY: Elsevier.

Schank, R. C. (1982). *Dynamic memory: A theory of reminding and learning in computers and people*. New York: Cambridge University Press.

Schank, R. C., & Abelson, R. P. (1977). *Scripts, plans, goals and understanding: An inquiry into human knowledge structures*. Mahwah, NJ: Lawrence Erlbaum Associates.

Shivashankar, V., Alford, R., Kuter, U., & Nau, D. S. (2013). The godel planning system: A more perfect union of domain-independent and hierarchical planning. *IJCAI* (p. 2380–2386).

Shivashankar, V., Kuter, U., Nau, D., & Alford, R. (2012). A hierarchical goal-based formalism and algorithm for single-agent planning. *Proc. of AAMAS* (pp. 981–988). Valencia, Spain.

Valmeekam, K., Marquez, M., Sreedharan, S., & Kambhampati, S. (2024). On the planning abilities of large language models: a critical investigation. *Proc. of NeurIPS*. Red Hook, NY, USA.

Wilensky, R. (1983). *Planning and understanding*. London: Addison-Wesley.

Zaidins, P., Goldman, R. P., Kuter, U., Nau, D., & Roberts, M. (to appear). HTN plan repair algorithms compared: Strengths and weaknesses of different methods. *Proc. of ICAPS*. Preprint available at: https://arxiv.org/abs/2504.16209.

Appendix A. Condensed OOMPA Restaurant Domain

This listing shows the OOMPA implementation of the Restaurant script for the entering and ordering scenes. It includes only the objects and roles necessary for those two scenes, though we have implemented other scenes as well.

```
class RestaurantDomain(AbstractDomain):
    def __init__(self) -> None:
        AbstractDomain.__init__(self, "restaurant")
        self.declare_types(Table, MenuItem, Dish, Menu, Person, Server, Patron)

    def test_create_simple_problem(self):
        problem = self.instantiate_problem()

        sam = problem.sam = Server("sam")
        menu = problem.menu_breakfast = Menu("menu_breakfast")
        special = problem.special = MenuItem("veggie_omelette", 12)
        menu.add_special(special)
        problem.pat = Patron("pat")
        problem.chris = Cook("chris")
        table1 = problem.table1 = Table("table1", server=sam, menu=menu)
        return problem


class MenuItem(AbstractNamed, HasStateProperties):
    class Status(StrEnum):
        AVAILABLE = auto()
        UNAVAILABLE = auto()

        def __str__(self):
            return self.name

        def __repr__(self):
            return self.__str__()

    cost: int = StatePropertyFactory()
    status: Status = StatePropertyFactory(Status.AVAILABLE)

    def __init__(self, name, cost: int, status: Status = Status.AVAILABLE):
        super().__init__(name)
        self.cost = cost
        self.status = status


class Menu(AbstractNamed, HasStateProperties):
    items: list[MenuItem] = StatePropertyFactory(default_factory=list)
    special: MenuItem | None = StatePropertyFactory(None)

    def __init__(self, name):
        super().__init__(name)

    def add_item(self, item: MenuItem):
        self.items.append(item)

    def add_special(self, special: MenuItem):
        self.special = special
```

```
class Table(AbstractNamed, HasStateProperties):
    server: Server = StatePropertyFactory()
    occupied: bool = StatePropertyFactory(False)
    menu: Menu | None = StatePropertyFactory(None)

    def __init__(self, name, server: Server, menu: Menu, occupied=False):
        super().__init__(name)
        self.server = server
        self.menu = menu
        self.occupied = occupied

    def init_add_menu(self, menu: Menu):
        self.menu = menu

    def init_add_patron(self):
        self.occupied = True


class OrderedItem(AbstractNamed, HasStateProperties):
    class Status(StrEnum):
        UNORDERED = auto()
        SELECTED_BY_PATRON = auto()
        ORDERED = auto()
        PREPARING = auto()
        PREPARED = auto()
        TOTALED = auto()

    PREFIX = "ordered_"

    patron: Patron = StatePropertyFactory()
    item: MenuItem = StatePropertyFactory()
    status: Status = StatePropertyFactory(Status.UNORDERED)
    dish: Dish | None = StatePropertyFactory(None)

    def __init__(self, patron: Patron, item: MenuItem):
        super().__init__(OrderedItem.PREFIX + item.name)
        self.patron = patron
        self.item = item


class Person(AbstractNamed): ...


class Server(Person, HasStateProperties):
    near_to: Person | None = StatePropertyFactory(None)
    order: OrderedItem | None = StatePropertyFactory(None)
    menu: Menu | None = StatePropertyFactory(None)
    check: Check | None = StatePropertyFactory(None)

    def __init__(self, name):
        AbstractNamed.__init__(self, name)
```

```
(cont) class Server(Person, HasStateProperties):

    # =======================================================
    # region action move_near

    @OompaAction
    def move_near(self, person: Person):
        pass

    @move_near.precondition
    def move_near(self, person: Person):
        return AndCondition(
            self.near_to.not_equals(person),
        )

    @move_near.effect
    def move_near(self, person: Person):
        return AndEffect(
            self.near_to.assigned(person),
        )

    # endregion action move_near
    # =======================================================


class Patron(Person, CreatesNewObjects, HasOompaMethods):
    problem: AbstractProblem
    money: int = StatePropertyFactory(50)
    is_hungry: bool = StatePropertyFactory(True)
    is_pleased: bool = StatePropertyFactory(False)
    table: Table | None = StatePropertyFactory(None)
    server: Server | None = StatePropertyFactory(None)
    menu: Menu | None = StatePropertyFactory(None)
    desired_order: OrderedItem | None = StatePropertyFactory(None)

    def __init__(self, name):
        AbstractNamed.__init__(self, name)

    # =======================================================
    # region action sit

    @OompaAction
    def sit(self, table: Table):
        pass

    @sit.precondition
    def sit(self, table: Table):
        return self.table.equals(None)

    @sit.effect
    def sit(self, table: Table):
        return AndEffect(
            self.table.assigned(table),
            self.server.assigned(table.server),
            table.occupied.assigned(True),
        )

    # endregion action sit
    # =======================================================

[cont] class Patron(Person, CreatesNewObjects, HasOompaMethods):

    # ========================================================
    # region action pickup_menu

    # TODO might also need put_down_menu, but we'll ignore it for now

    @OompaAction
    def pickup_menu(self):
        pass

    @pickup_menu.precondition
    def pickup_menu(self):
        return AndCondition(
            self.table.menu.not_equals(None),
            self.menu.equals(None),
        )

    @pickup_menu.effect
    def pickup_menu(self):
        return AndEffect(
            self.menu.assigned(self.table.menu),
        )

    # endregion action pickup_menu
    # ========================================================


    # ========================================================
    # region action review_menu

    @OompaAction
    def review_menu_for_special(self): pass

    @review_menu_for_special.precondition
    def review_menu_for_special(self):
        return AndCondition(
            self.menu.not_equals(None),
            self.desired_order.equals(None),
        )

    @review_menu_for_special.effect
    def review_menu_for_special(self):
        return AndEffect( InsertNewObjectEffect(
                self,
                self.desired_order,
                OrderedItem,
                [self, self.menu.special],
                self.problem,
                {}, ),
            self.desired_order.status.assigned(OrderedItem.Status.SELECTED_BY_PATRON),
        )

    # endregion action review_menu
    # ========================================================
[cont] class Patron(Person, CreatesNewObjects, HasOompaMethods):
    # ========================================================
    # region action request_server

    @OompaAction
    def request_server(self):
        pass

    @request_server.precondition
    def request_server(self):
        return AndCondition(
            self.server.near_to.not_equals(self),
        )

    @request_server.effect
    def request_server(self):
        return AndEffect(
            self.server.near_to.assigned(self),
        )

    # endregion action request_server
    # ========================================================


    # ========================================================
    # region action place_order

    @OompaAction
    def place_order(self):
        pass

    @place_order.precondition
    def place_order(self):
        return AndCondition(
            self.desired_order.not_equals(None),
            self.server.near_to.equals(self),
        )

    @place_order.effect
    def place_order(self):
        return AndEffect(
            self.server.order.assigned(self.desired_order),
            self.desired_order.status.assigned(OrderedItem.Status.ORDERED),
        )

    # endregion action place_order
    # ========================================================
[cont] class Patron(Person, CreatesNewObjects, HasOompaMethods):

    # =======================================
    # region Method m_entering
    @OompaMethod
    def m_entering(self, table: Table) -> GoalMethod:
        pass

    @m_entering.goal
    def m_entering(self, table: Table) -> Condition:
        goal = self.table.equals(table)
        return goal

    @m_entering.precondition
    def m_entering(self, table: Table) -> Condition:
        return AndCondition( table.occupied.equals(False),
                             self.table.equals(None), )

    @m_entering.body
    def m_entering(self, table: Table) -> TotalOrderGoalTaskNetwork:
        return TotalOrderGoalTaskNetwork(
            self.sit(table),
        )

    # endregion Method m_entering
    # =======================================

    # =======================================
    # region Method m_ordering
    @OompaMethod
    def m_ordering(self, table: Table) -> GoalMethod:
        pass

    @m_ordering.goal
    def m_ordering(self, table: Table) -> Condition:
        return AndCondition( self.desired_order.not_equals(None),
                             self.desired_order.status.equals(OrderedItem.Status.ORDERED))

    @m_ordering.precondition
    def m_ordering(self, table: Table) -> Condition:
        return AndCondition(
            self.desired_order.equals(None),
            table.menu.not_equals(None),
        )

    @m_ordering.body
    def m_ordering(self, table: Table) -> TotalOrderGoalTaskNetwork:
        body = TotalOrderGoalTaskNetwork(
            self.sit(table),
            self.pickup_menu(),
            self.review_menu_for_special(),
            self.request_server(),
            self.place_order(),
        )
        return body

    # endregion Method m_ordering
    # =======================================
```