 Nik Boyd

State Names

Naming Conventions

Intent

Consistently name the states that form complex finite state machines (FSMs).

Motivation

The State pattern[1] allows you to model the states of finite state machines (FSMs) with objects. During usage, an object alters its behavior when its internal state changes, thereby appearing to change its class.

The State pattern has several useful positive consequences:

- *It partitions and localizes behavior for different states.* You can separate all the behavior that's specific to each particular state into each state object. New states and transitions can then be introduced more easily. In contrast, if you put the behavior for all states into a single object, large conditional statements will typically result.
- *It makes state transitions explicit.* Instead of using internal data values, which define states implicitly, states are identified explicitly and the transitions between states are both explicit and atomic.
- *State objects can be shared.* State objects are Flyweights,[1] because they have no instance variables, and they identify the states they represent entirely by their type. Thus, they provide only behavior, which can be safely shared by multiple threads.

Because the State pattern uses an object for each machine state, each state object needs a class name. Thus, naming conventions for machine states are important, especially for extensive FSMs with many states.

However, apart from those implied by a simple example, the State pattern[1] offers little guidance on naming machine states (and hence the objects that represent those states). In contrast, the State Names pattern offers specific recommendations for naming FSM states based on the concepts of operational *scenarios* and state *sequencing*.

Applicability

Use the State Names pattern when:

- you want to use the State pattern for modeling operational scenarios (simple or complex)
- you want to use the State pattern for modeling language grammars

Simple Operational Scenario Naming Conventions

An object with a simple lifecycle can be defined with a simple FSM. Simple FSMs often have fairly obvious state names. In a simple FSM, the state names often follow the object operations, and the state names can have a fairly obvious relationship to the operational verbs used in the object operation names. Thus, each state in a simple FSM can be named using the past tense of each transition operation (transitive verb) that immediately precedes the state.

> *For simple operational FSMs, name the machine states using descriptive adjectives that directly correspond to the state transition operations, i.e., use the past tense of the transitive verb for each state transition operation.*

Consider the example offered by the State pattern.[1] A TCP connection scenario transitions through only a few states that correspond to (and follow after) the connection operations, including establishing, listening on, and closing a connection. Thus, the State pattern example uses a simple naming pattern:

Context = *Entity* + *Scenario*

A context class name identifies an *entity* and its state machine *scenario*. For example, TCPConnection.

Concrete State = *Entity* + *Operation* (+ ed)

Each concrete state name identifies the entity and the operation that leads (transitions) to the subject machine state. For example, TCPEstablished, TCPListen, TCPClosed.

Complex Operational Scenario Naming Conventions

More complex operational state machines require more elaborate naming conventions, especially non-deterministic machine scenarios for conducting interactions with other agents (like a user). Consider a cash dispenser within an automated teller machine (ATM). While an ATM is often used as an example for object-oriented analysis exercises, such analyses are often superficial, and they usually overlook important details. Correctly dispensing cash is a primary value of ATMs. So, getting the details of this scenario correct is fairly important.

Cash dispensers are mechanical devices that dispense bills. Simple dispensers offer bills of only a single denomination, while more advanced dispensers offer a mix of bills (of various denominations and perhaps even various currencies). As mechanical devices, they are subject to failures throughout the path that bills travel. Thus, a scenario for dispensing cash will typically entail several mechanical operations tied together by bill transport sensor readings. The transport sensors detect the presence of bills in the various stages of the dispense path and determine the appropriate sequence of mechanical operations (transitions between states). Figure 1 depicts a typical state model for dispensing cash from a cash dispenser.

Note that some mechanical operations may appear more than once, but in different parts of the state model. Also, note the loops that appear in the model prior to bills arriving at the dispenser gate. Cash dispensers are fairly reliable. However, once a customer has requested cash, a dispenser should always fulfill a dispense request unless the dispenser has:

a. exhausted its bill supply, or
b. suffered an unrecoverable mechanical failure.

Also, it has been observed that bank customers sometimes get distracted, even when withdrawing funds from an account. So, after dispensing cash, the dispenser verifies that the bills are subsequently removed from the dispenser gate. If the bills have not been removed after a specific duration (configurable timeout), the dispenser retracts the bills from the gate and deposits them into an internal bin for recovery by a bank employee. The customer account will subsequently be credited with the amount recovered.

Applying the State pattern to such a FSM requires a naming convention that captures the additional complexity of the state model. Complex state machines often include loops and possibly repetitive use of some operations. So, the proper sequencing of operations becomes an important part of the state model.

For complex operational FSMs, name the machine states using prepositional phrases that describe operations performed before, during, and after a main success scenario (MSS).

The machine state naming conventions used in Figure 1 can be summarized as follows:

Context = *Entity* + *Scenario*

A context class name identifies an *entity* and its state machine *scenario*. For example, ATMCashDispense.

Antecedant State = *Operation* + Before + *EntityScenario*

Each antecedant state identifies an operation used before a main success scenario. For example, ClearPathBeforeCashDispense.

Main Scenario State = *Operation* + During + *EntityScenario*

Each main scenario state identifies an operation used during a main success scenario. For example, PresentBillsDuringCashDispense.

Subsequent State = *Operation* + After + *EntityScenario*

Each subsequent state identifies an operation used after a main success scenario. For example, RetractBillsAfterCashDispense.

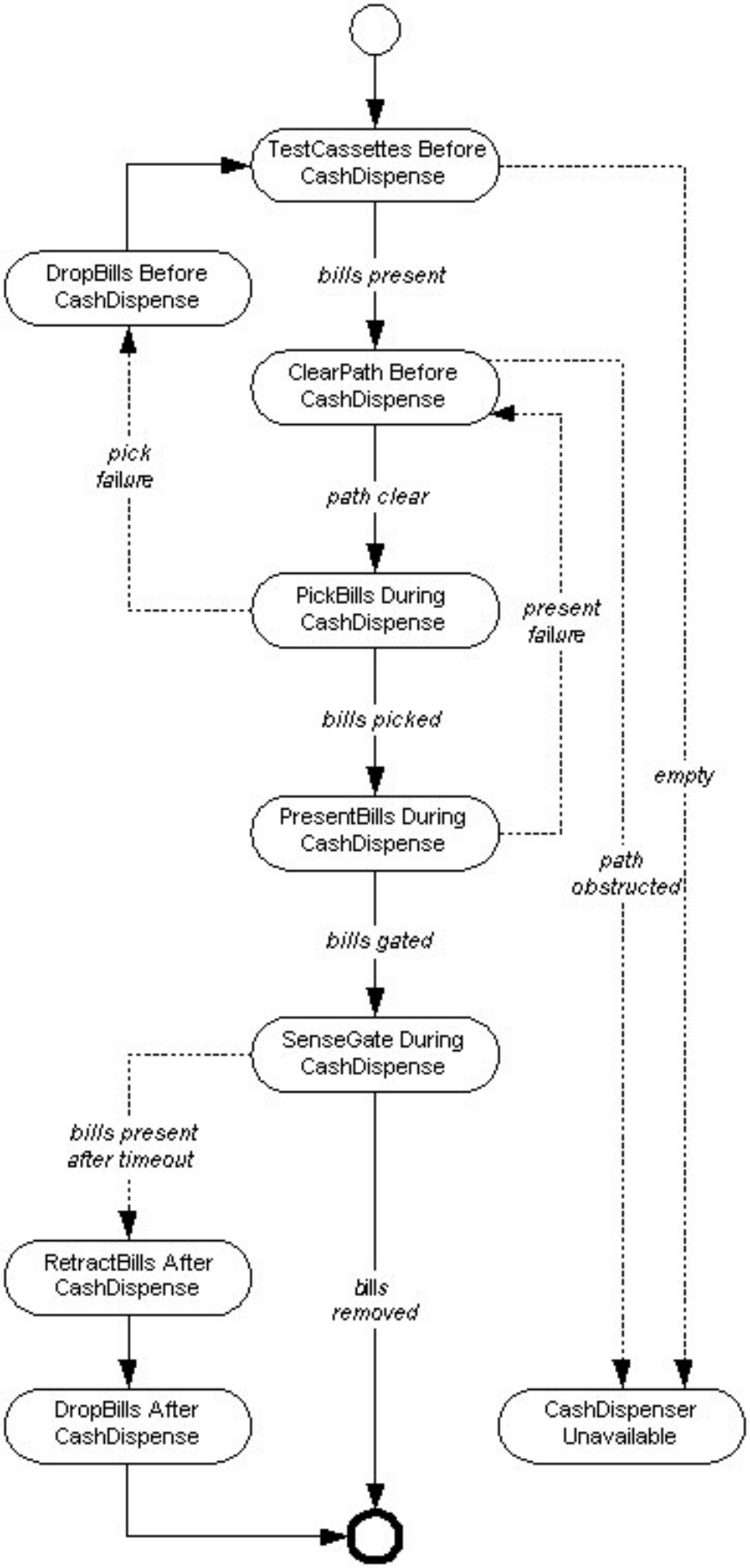

Figure 1. Cash Dispense Scenario

Language Grammar Naming Conventions

Consider a grammar that defines a programming language. Programming language grammars are among the most complex of state machines, often including nearly (or in excess of) 100 states. Such grammars are often defined in terms of productions, each of which describes a scenario (a sequence of grammatical terms) that produces a result.

A few example productions excerpted from a programming language grammar offer a representative sample of the kinds of sequences included in such grammars.

VariableDeclarations = (VariableDeclaration)+

VariableDeclaration = VariableOptions VariableName
TypeAnnotationOption InitializationOption **'.'**

TypeAnnotationOption = (TypeAnnotation)?

TypeAnnotation = **'('** TypeName **')'**

TypeName = PackageNameOption FaceName

PackageNameOption = (PackageName)?

PackageName = Identifier PackageSuffixOptions

PackageSuffixOptions = (PackageSuffix)*

PackageSuffix = **'.'** Identifier

VariableOptions = AccessChoiceOption FinalOption

AccessChoiceOption = (**'public'** | **'protected'** | **'private'**)?

FinalOption = (**'final'**)?

FaceName = Identifier

VariableName = Identifier

Identifier = Letter LetterDigitChoiceOptions

LetterDigitChoiceOptions = (LetterDigitChoice)*

LetterDigitChoice = (**Letter | Digit**)

BinarySelector = OperatorChoices

OperatorChoices = (**OperatorChoice**)+

Because of its good correspondence for the recommended naming conventions, the grammar notation used in this example comes from ANTLR[2] rather than the more commonly used EBNF. However, the suggested naming conventions are generally applicable across grammars irrespective of the notation used to define them. Also, in order to better show the kinds of state naming conventions available, these example productions have been decomposed beyond what would usually be presented in a grammar summary.

> *For highly complex FSMs described by context free grammars (CFGs), name the machine states using fully decomposed grammar productions, combined with appropriate usage of choices and options.*

The machine state naming conventions used in the sample grammar can be summarized as follows:

(*State*)+ = *State* + s
: For a state that occurs repeatedly in a sequence, simply pluralize the state name. For example, VariableDeclarations was derived from VariableDeclaration.

(*State*)? = *State* + Option
: For an optional state, simply append the word Option to the state name. For example, TypeAnnotationOption was derived from TypeAnnotation.

(*State*)* = *State* + Options
: For an optional state that may appear repeatedly in a sequence, simply append the word Options to the state name. For example, PackageSuffixOptions was derived from PackageSuffix.

(*StateA* | *StateB* | ...) = *Selection* + Choice
: For a state in which an alternative must be chosen, append the word Choice to the name of the selection. For example, LetterDigitChoice was derived from the legal identifier characters.

(*StateA* | *StateB* | ...)+ = *Selection* + Choices
: For a state in which some alternative must be chosen repeatedly, append the word Choices to the name of the selection. For example, OperatorChoices was derived from OperatorChoice.

(*StateA* | *StateB* | ...)? = *Selection* + Choice + Option
: For a state in which an alternative may be chosen, append the words Choice Option to the name of the selection. For example, AccessChoiceOption was dervied from the available access modifiers.

(*StateA* | *StateB* | ...)* = *Selection* + Choice + Options
: For a state in which some alternative may be chosen repeatedly, append the word Choice Options to the name of the selection. For example, LetterDigitChoiceOptions was derived from LetterDigitChoice.

Consequences

To use the State Names pattern effectively, you may need to:

- decompose a complex FSM into simpler machines
- combine the various State Name conventions

Decomposition is one of the natural benefits of working with CFGs. Indeed, each production in a CFG can be viewed as a small FSM. Thus, CFGs allow you to decompose the structure of valid sequences until you can apply one or more of the naming conventions.

References

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing, Inc., 1995. ISBN 0-201-63361-2.
2. Terrence Parr. ANTLR Parser Generator and Translator. http://www.antlr.org/
