Unit 1

Brickles Specifications and Analysis Model

Contents

In this unit we present the specifications for *Brickles*, an arcade game to be hosted on a personal computer operating under Windows. Next, we present an object model for the game that shows the structure of the problem—the object classes, the relationships between them, and some of the responsibilities. The focus of this course will be in exposing how a design is produced from an analysis model and how a design is implemented in C++. Consequently, it is important to understand object models. We will use models expressed in the "Unified Method" notation in this course. [Please note that other notations exist, but the notation we use is widely accepted.]

You will be given an opportunity to play *Brickles* to help you gain an understanding of the requirements.

Contents

- Requirements of *Brickles*, the arcade game used in this course
- Unified notation for object models and state diagrams
- An analysis model for *Brickles*.

Requirements

The following pages present a description of the requirements for *Brickles*. The description presents the significant behaviors associated with playing the game.

Requirements

Overview

The game of Brickles includes a puck which moves continuously across a rectangular playing field that is bounded on all four sides. If we picture the field as being vertical, i.e. as it appears on a computer screen then we can think of the four sides as two walls, a ceiling, and a floor.

As the puck moves, it may encounter various obstacles. Besides the boundaries of the field described above, these include a paddle and a number of bricks. The paddle resides near the floor and the bricks near, but not touching, the ceiling.

When the game starts, the puck begins to move in a downward direction. It is up to the player of the game to move the paddle so that the puck hits the paddle and not the floor. The puck then reflects upward. Whenever the puck hits a brick, the brick is destroyed. The object of the game is to destroy all the bricks.

The paddle is controlled by the player. One objective of the game is for the player to keep the puck "in play" by moving the paddle into position to collide with the puck and to send it rebounding into the playing field. The second objective is to position the puck so that it collides with and destroys the bricks.

The playing field of the game sits within an environment that includes the means for controlling the flow of the game. For example, the environment provides the means for exiting the game.

The player is given a specified number of pucks with which to destroy the bricks. Winning the game means that the player has eliminated all bricks without running out of pucks. The player loses if the supply of pucks is exhausted before all of the bricks are destroyed.

High-Level Use Cases

There is one major actor in this system, the Player.

The player is involved in the actual play of the game. In particular, this actor is responsible for:

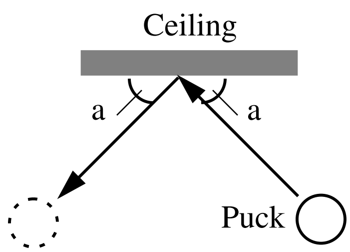
- running and ending the game application,
- starting, pausing, resuming, and ending a match, and
- moving the paddle to keep the puck in play.

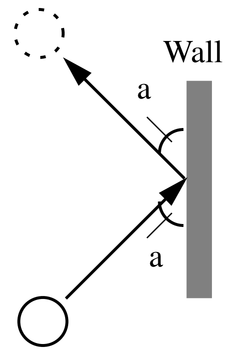
Requirements (continued)

Specifics

Ceiling and walls

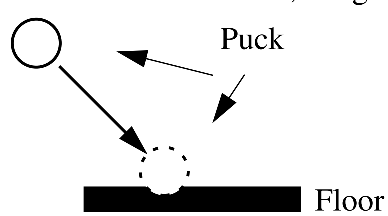
The puck bounces off the ceiling and walls in accordance with the laws of physics, neglecting friction and gravity i.e. angle in equals angle out.





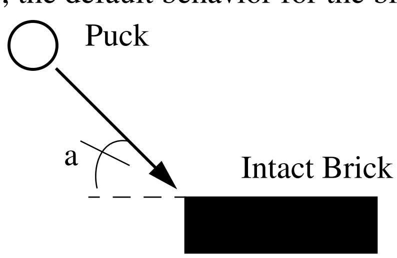
Floor

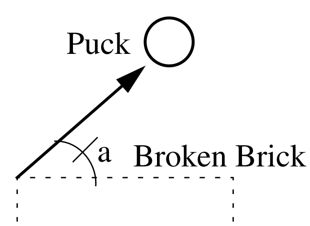
The floor acts like a tar pit as far as the puck is concerned. If the puck hits the floor it does not rebound. That puck is considered lost. The player is given a fixed number of pucks at the beginning of the game. Once those pucks have all hit the floor, the game is over and the player loses.



Bricks

Bricks reflect the puck in the same way that the ceiling does - angle in equals angle out. Note that unlike the ceiling, bricks can be hit from above as well as from below. They also have sufficient thickness that they could be hit from the side For the sake of simplicity we will assume that bricks are treated like horizontal objects regardless of which part of the brick is struck by the puck. Thus it is the vertical component of the puck's direction that is changed when the puck hits the brick. Finally, the default behavior for the brick upon being struck is that the brick is destroyed.





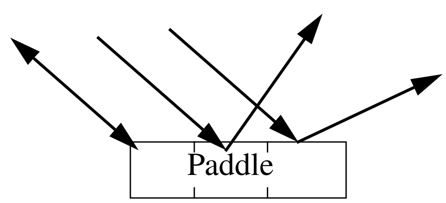
Requirements (continued)

Paddle

The paddle is the most interesting obstacle in the game. It provides the only means for the player to control the direction of the puck. Note that if the player had no control - if the puck bounced off the paddle strictly based on the laws of physics - then it could become very difficult, perhaps even impossible, to destroy the last few bricks.

The rules for the paddle are based on the direction of the puck as it hits the paddle and on which part of the paddle is hit. If we imagine the paddle has been divided into thirds we can define the *near third* as being the left third of the paddle if the puck is coming in from the left and the right third if the puck is coming in from the right. We can similarly define the *far third,* and the phrase, *middle third,* is self explanatory. The rules may be summarized as follows:

- If the puck hits the paddle on its near third then the puck returns in the exact opposite direction from which it came.
- If the puck hits the paddle on the middle third then the reflection is almost that of a typical horizontal object except that, up to some limit, the outgoing angle is a little steeper than the incoming angle. We need the limit because we don't want the puck to go completely vertical.
- If the puck hits the paddle on the far third then the reflection is almost that of a typical horizontal object except that, up to some limit, the outgoing angle is a little shallower than the incoming angle. This time we need the limit because we don't want the puck to go completely horizontal.



Puck

The player is given a fixed number of pucks at the beginning of the game, but only one is in play at any given time. Once one pick hits the floor, the next is brought into play, assuming there are any more available. The puck has a current direction and speed and moves according to an automatic timer. As seen above, collisions may change the direction of the puck but not the speed.

Requirements (continued)

So far we have only discussed the playing field of the game and its contents. The game is played in a Windows environment.

Environment

The playing field resides within a larger environment that shall include the following:

- A means for starting and pausing the match. When the match is paused, it should not be possible to move the paddle.
- A means to exit the game before a match is ended.
- A message displaying the number of pucks remaining.
- A congratulatory message in case the player wins the game and a consolation message in case the player loses the game.

Exercise

Experience the game

- Spend some time playing the game to understand the requirements.
- Try each of the controls—the mouse and left button—to see their effects.
- Note the different sequences of bounces that the puck takes.

Analysis—*What* is the problem?

An object-oriented development effort includes analysis—the process of understanding *what* the problem is, including the requirements.

Object-oriented analysis focuses on three aspects of objects:

- Static structure (object model): What are the object classes? What are their relationships? How are they organized?
- Sequencing of interactions (dynamic model): What is the time-dependent behavior of the system and the objects in it?
- Data transformations (functional model): How do outputs get computed from inputs?

Each model describes one aspect of the system. Each will contain references to the contents of other models.

NOTE: We will focus on *application analysis* in this course—that is, we will focus specifically on the object classes and relationships that exist in *Brickles*. Many efforts include *domain analysis*, which focuses first on the context of an application—in this case, on arcade games—in an effort to identify more abstract classes and relationships.

Analysis—*What* is the problem?

Object-oriented analysis focuses on objects to gain an understanding of a problem:

- Object models capture the static structure of the problem:
  - Classes
  - Relationships
  - Organization
- Dynamic models capture the sequencing of actions in a system, including the behaviors associated with objects
- Functional models capture processing requirements—the transformation of inputs to outputs

Object Models

"An object model describes the structure of objects in a system—their identity, their relationships to other objects, their attributes, and their operations." [Rumbaugh et al]

- How is an object model constructed?
- Identify object classes
- Prepare class descriptions
- Identify relationships
- Identify associations (and aggregations) between objects
- Classify objects
- Identify responsibilities
- Verify the model

Note that these activities are performed iteratively to refine the model. In addition, a model can be organized into sheets and into subsystems (a.k.a. categories).

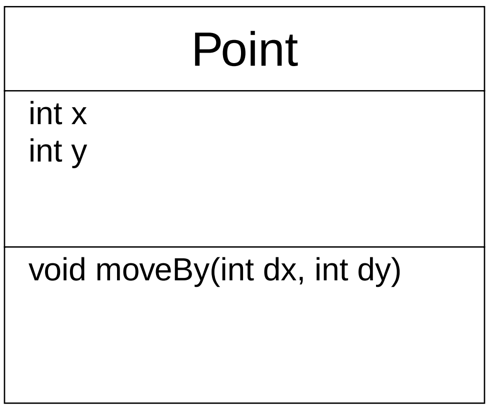
The two primary components of an object model are object classes and associations. The notations for each are shown.

NOTES:

- Construction of an object model is beyond the scope of this course.
- The unified notation contains a much richer set of symbols for the expression of objects and relationships than are presented in these notes. We present only those needed for *Brickles*.

Object Models

- Object Classes: A class describes a set of objects with similar properties (attributes), common behavior (operations), common semantics, and common relationships to other objects. Notation:



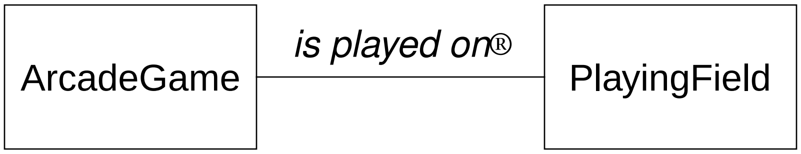
class name

attributes—data values held by the objects in a class

operations (and methods)—a function or transformation that may be applied to or by objects in a class

- Associations
  - A *link* is a physical or conceptual connection between object instances. It shows a relationship between two or more *objects*.
  - An *association* describes a group of links with a common structure and common semantics. Note that an association is inherently bidirectional. Notation for an association:

Binary association:



Multiplicities of Associations

Since an association describes a group of links, it is important to understand how many objects of the class at one end of an association can be linked (at one time) to a single object at the other end of the association. Each end of an association is annotated with a *multiplicity*:

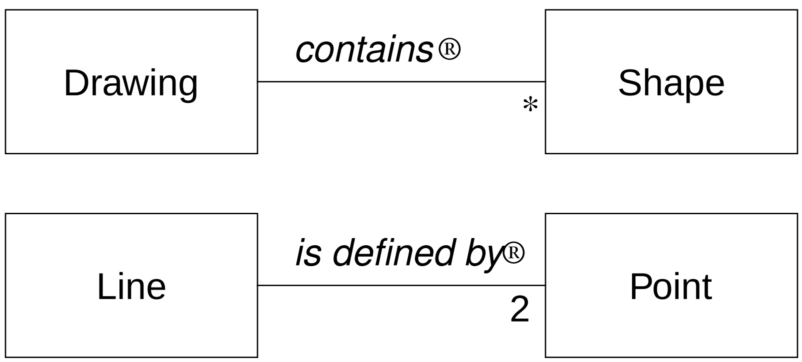
- A given workstation can optionally display a console, as indicated by "0,1". Conversely, a console is always associated with exactly one workstation, as indicated by no annotation, which implies "1".
- A given drawing object can contain zero or more shapes, as indicated by "*," a symbol used in language theory to mean "zero or more". [Note that the symbol "+" indicates "one or more".] Conversely, a given shape is contained in exactly one drawing.
- A given line is defined by exactly two points. Conversely, a given point is used to define exactly one line. [This model, therefore, precludes two lines being defined by a given point.]
- A given student is signed up for zero or more sections of a course at a time, and a given section can have zero or more students signed up for it.

Multiplicity of Associations

- *Multiplicity* specifies how many instances of one class may relate to a single instance of an associated class.

Optional:

One-to-many:



Many-to-many:

Aggregation

Aggregation is a special kind of association that models a part-whole relationship. It is indicated by the use of a diamond, drawn at the end of the association that names the "larger" object. Aggregation is distinguished from other kinds of associations because it possesses three special properties:

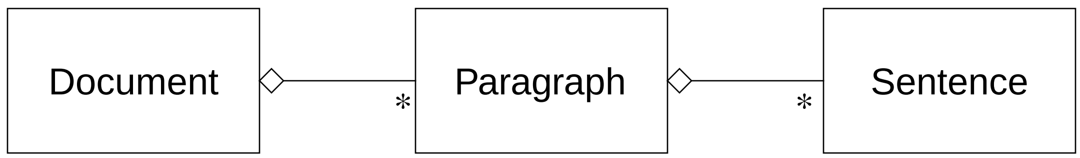
- Transitivity. If a sentence is part of a paragraph, and a paragraph is part of a document, then a sentence is part of a document.
- Antisymmetry. If a paragraph is a part of a document, that document is not part of that paragraph.
- Propagation of properties. If a paragraph is part of a document and the page margins for the document are changed, then the margins of the paragraph are also changed.

The last property is sometimes hard to establish. If in doubt, don't use aggregation in the model.

NOTE: The meaning of aggregation is generally determined by consensus among a group of developers. One commonly accepted property is that the lifetime of object in an aggregation corresponds to the lifetime of the aggregate itself—for example, if a document is destroyed, then all the paragraphs (and sentences) in it are destroyed as well.

Aggregation

- *Aggregation* is a special kind of association describing a part-whole (*is part of*) relationship—for example, a bill of materials or a parts explosion tree.



- Properties of aggregation are:
  - *Transitivity:* if A *is a part of* B and B *is a part of* C, then A *is a part of* C.
  - *Antisymmetry:* if A *is a part of* B, then B *is not a part of* A.
  - *Propagation of properties:* if A *is a part of* B, then some of the properties of A are obtained from B.

  [When in doubt, use ordinary association.]

Generalization

Generalization is a relationship between classes—frequently referred to as the *is a* or *is a kind of* relationship. The notation for generalization is a triangle (arrowhead) that points to the more general class. In the example, a circle is a kind of shape, a rectangle is a kind of shape, and so on.

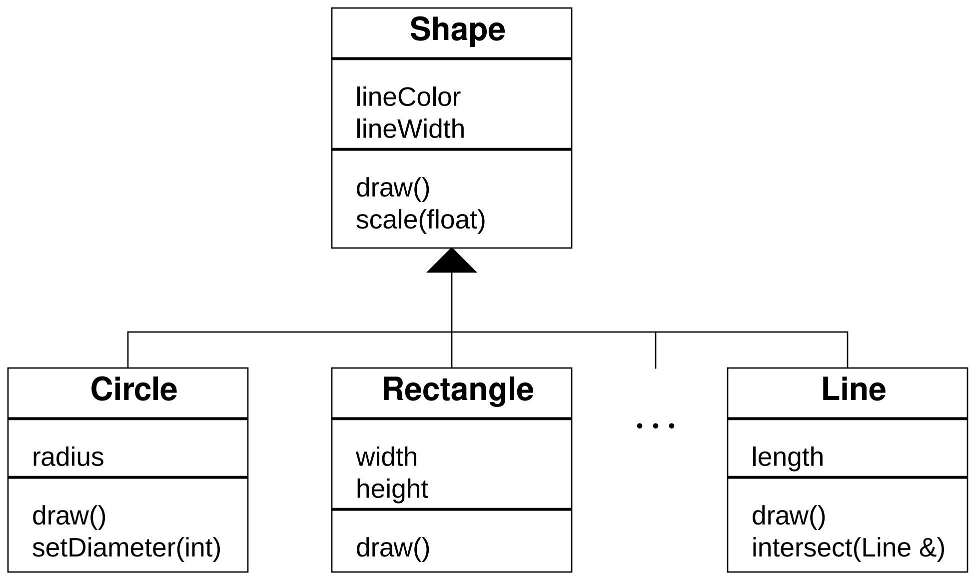
In analysis, we frequently look for commonalities among different classes of objects and model them using *abstract* classes—classes for which no direct instances exist, but that still characterize attributes, operations, and relationships. Any abstract class in an application analysis model should have at least one *specialization* that is *concrete*.

The semantics of generalization are similar to those of [public] inheritance in object-oriented programming languages: the specialized class—"subclass" or "derived class"—inherits all of the attributes and operations defined for the general class—"superclass" or "base class". The subclass can add more attributes, add more operations, or override operations—that is, change the way that operation works.

We will use generalization only for the *is a* relationship, and not as another name for inheritance—that is, not as a way to share code between two classes. We will sometimes refer to one class **D** as a *subtype* of a class **B**, meaning that all the operations defined for **B** are defined for **D**. When **D** is a subclass of **B**, then we not only expect **D** to be a subtype of **B**, we also expect the operations in both classes to have the same semantics and we expect the objects in class **D** to be a subset of the objects in class **B**.

Generalization

- *Generalization* is the relationship between a class and one or more refined versions of it. It is referred to as the *is a* or *is a kind of* relationship:



- Recall that classes can be either *abstract* or *concrete*. These hierarchies make abstract classes useful for setting up layers of abstraction.
- If class *B* is a generalization of class *D,* then

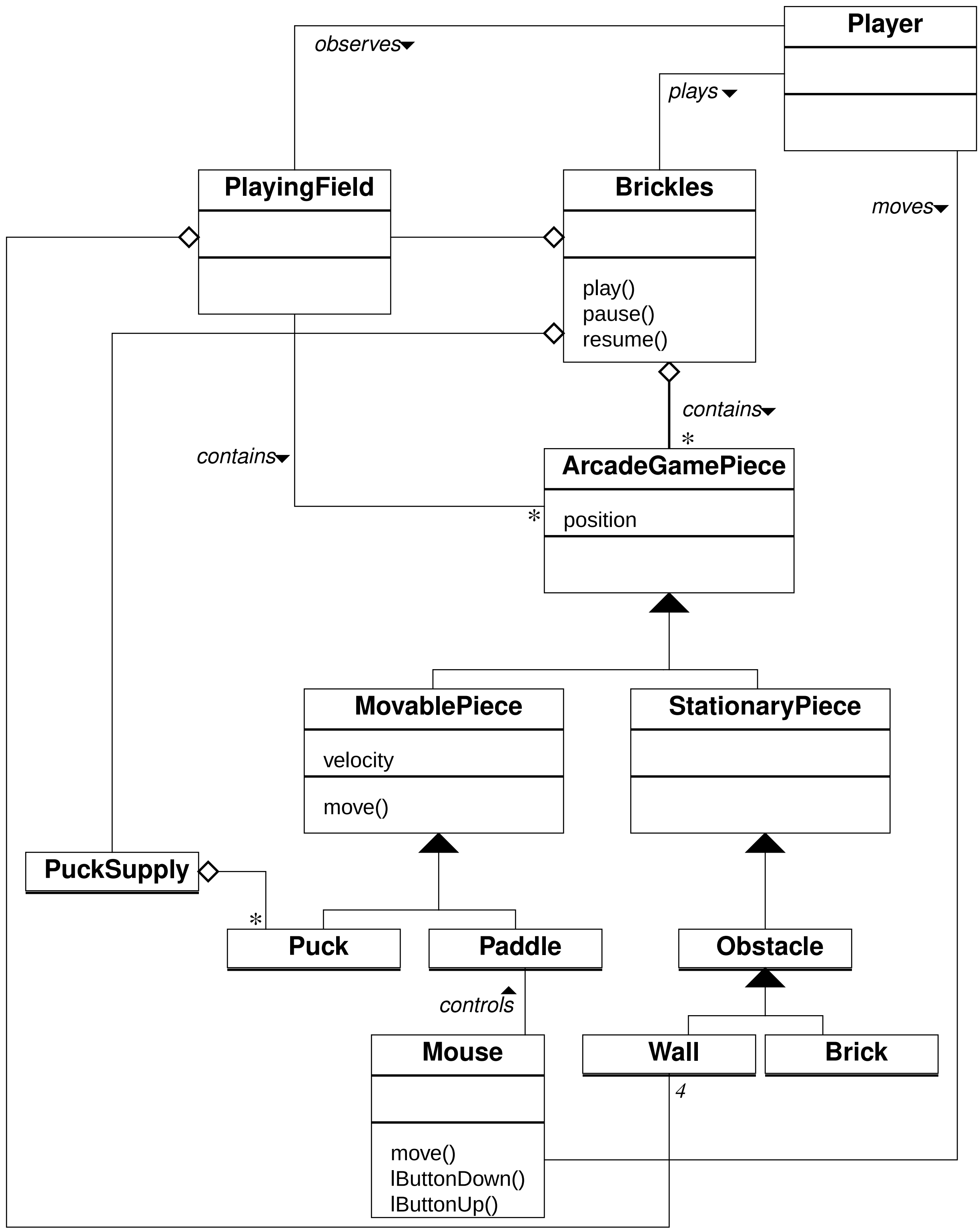
$$D \subseteq B$$
$$behavior(B) \subseteq behavior(D)$$

Brickles Analysis Object Model

The Brickles object model comprises the objects that make up the game: a playing field, a puck supply, and various arcade game pieces—pucks, bricks, walls, and paddles. The model does not attempt to model Brickles in great detail—for example, that a puck supply contains from zero to five pucks or a specific number of bricks—because the analysts felt that the users might want some changes in that area in the future. [One weakness of traditional systems analysis is its focus on a specific problem, which subsequently results ina system design that is not flexible or extensible.]

Of course this is only one possible model. An object model represents one view of the problem and is apt to be refined over the lifetime of a development effort.

Brickles Analysis Object Model



State Diagrams

A dynamic model addresses the time-related behavior of a system, where "time" includes the arrival of events. A state diagram for each class is an important component of a dynamic model.

A state diagram relates events and states. An *event* is something that happens at a point in time. Conceptually, it is "instantaneous." A *state* is an abstraction of the attribute values and links of an object, with sets of values being grouped together according to properties that affect the gross behavior of the object.

Within a state diagram, a state is represented by a "rounded" rectangle. Each state is labeled with a name. Transitions between states are indicated by arcs, optionally labeled by an event that triggers the transition. If no label appears, then the transition is automatic.

A diagram might show objects with finite lives. Instantiation is designated by a special start state (a solid circle), and deletion is designated by a bull's-eye.

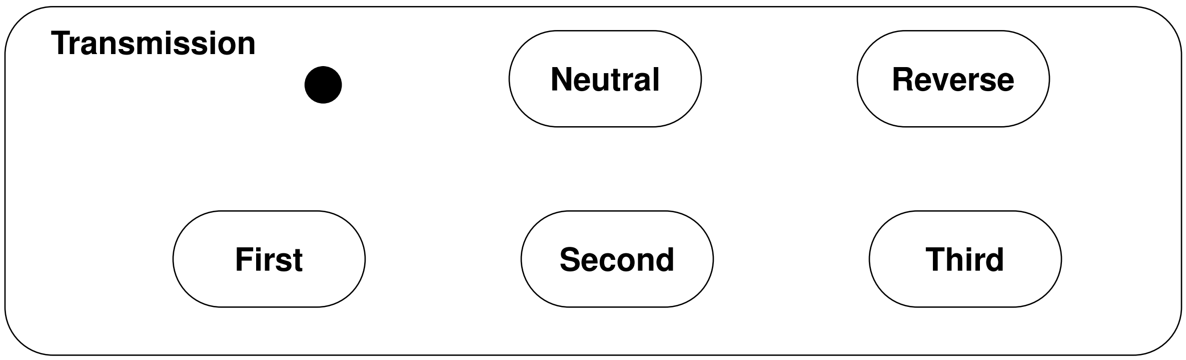
State diagrams may be nested, providing a form of generalization of states.

NOTE: The unified notation has additional syntax for expressing concepts in dynamic models—event attributes, conditions, and so on. We have presented only the syntax sufficient for our purposes.

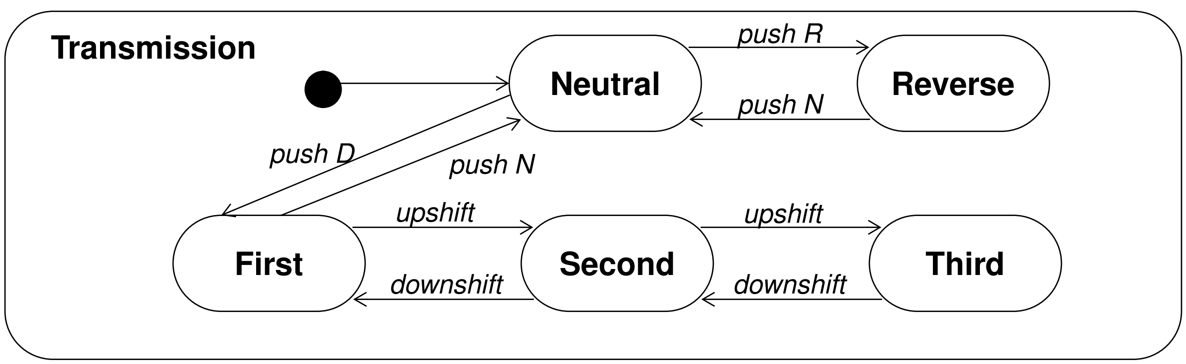
State Diagrams

A state diagram defines the states of an instance of a class and the transitions between them associated with various events.

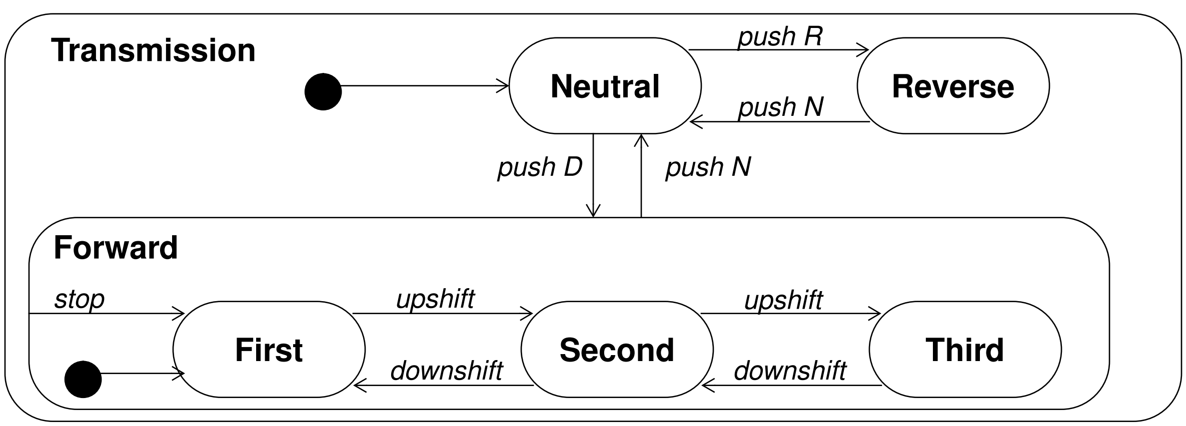
- States:



- Events and transitions:



- Nested states:



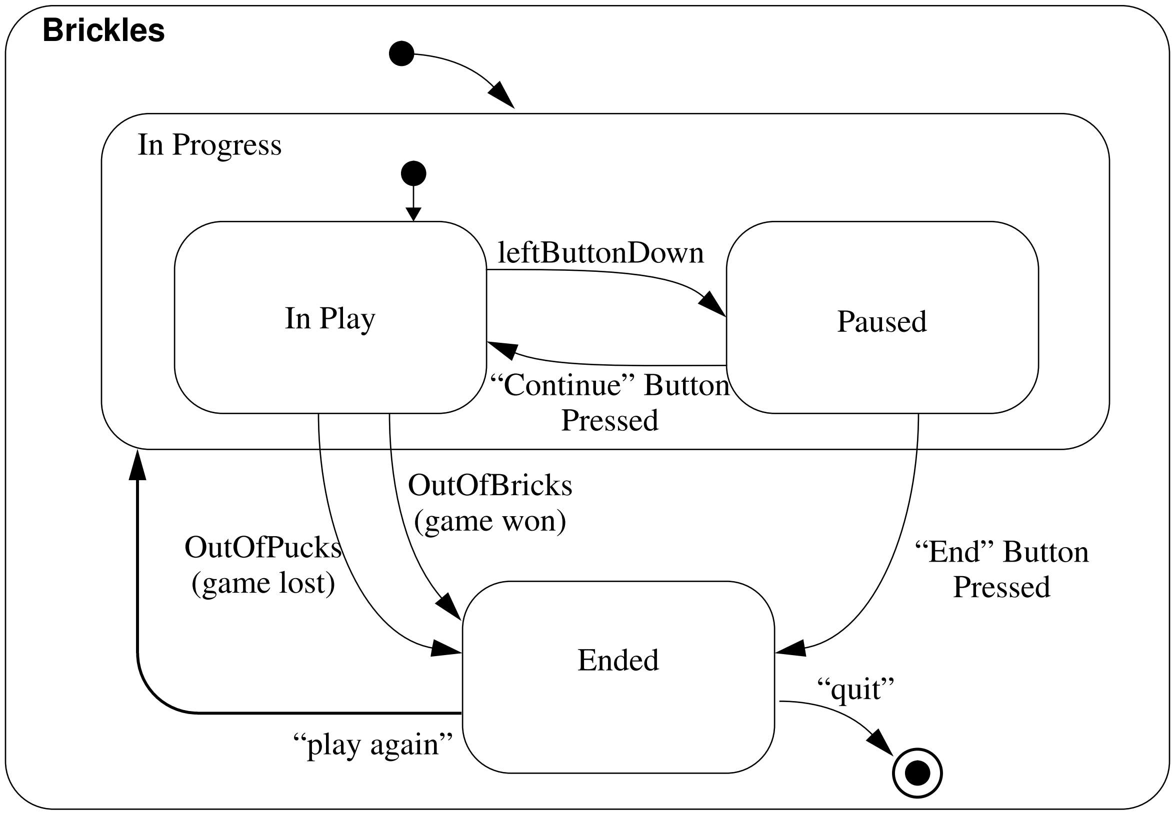
State Diagrams for Brickles

State diagrams for two of the classes in the object model for Brickles are shown.

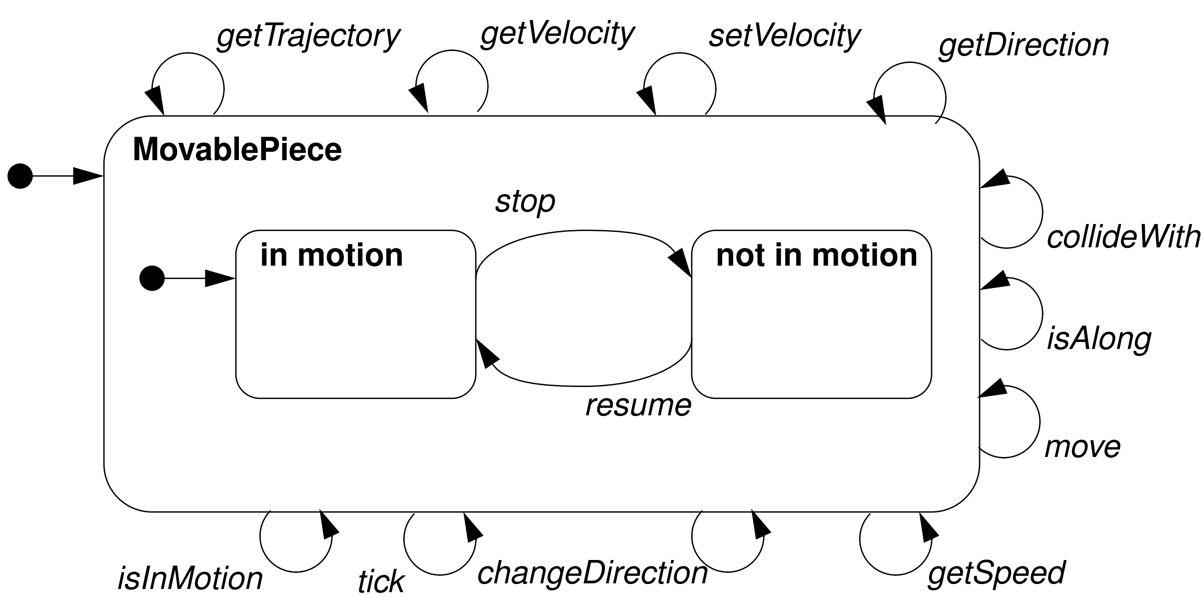
The first diagram shows the states associated with playing a game of Brickles. Note that a game is considered to be "in progress" as long as it is not ended. Can you see any states that might be added?

The second diagram shows the behavior of a movable piece. Note that many operations can be applied at any time, and that "move" is meaningful even if an instance is not in motion. A state diagram describes the behavior associated with an object over time—as measured, in part, by the arrival of events—but does not describe complete semantics associated with an operation. For example, what happens when a "move" is performed on a movable piece that is not in motion? [For the answer to that, we really have to look at documentation for the move operation.]

State Diagrams for Brickles



•



Summary

Take a few minutes to review the material in this unit. Note the three most important concepts from your perspective.

-
-
-

Summary

- The requirements for *Brickles* have been presented and you have had a chance to play the game to learn more about the behavior of the application.
- An analysis model has been presented to define the object classes in the problem, some of their attributes and behaviors, and their relationships with one another.
- An object model shows static relationships between objects by documenting associations and generalizations between classes.
- A state diagram, part of a dynamic model, shows the dynamic behavior of an instance of a class in terms of states and events.