Unit 3

Brickles Design—Increment 1

Topics

In this unit we begin to address the design of the *Brickles* application.

We will introduce a few basic concepts about computer graphics that will be used in the design. Then we will look at taking the analysis model into design. Our focus will initially be on a simple version of the game that addresses only the playing field, the puck, and the paddle. Once the simple version works, we will move on to developing the full game.

Topics

- Some basic computer graphics and physics concepts
- Critique of an existing Brickles design
- Goals for “Increment 1” of Brickles
- A design for Increment 1
- Implementation of part of Increment 1 (as an exercise)

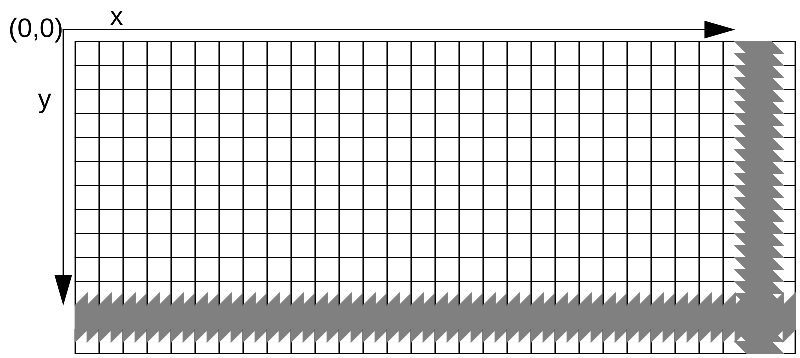
Basic Graphics

Within the windows environment, the screen is viewed as a rectangular array of pixels, each of which can display one of various colors. The coordinate system used has a different orientation from the one you probably learned in algebra class.

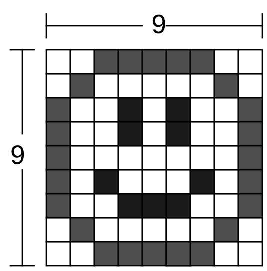
The figures that we draw on the screen—for the puck, the paddle, the bricks and so on—are represented as bitmaps (or, more correctly, “pixmaps”) that are themselves arrangements of pixels. Fortunately, we will not have to get involved with how a bitmap gets placed on the screen [it’s in the view, already implemented for us in such a way that when we put an object on the playing field, it’ll show up in the view]. We will be interested, however, in using bounding rectangles for objects that are drawn on the playing field. Bounding rectangles provide a mechanism for seeing if two objects overlap on the playing field. If their bounding rectangles intersect, then the objects overlap. The class CRect in MFC will be used to represent bounding rectangles. A CRect instance stores the top left and bottom right coordinates of a rectangle. Associated with a rectangle is also an extent: its length in the x and y directions.

Basic Graphics

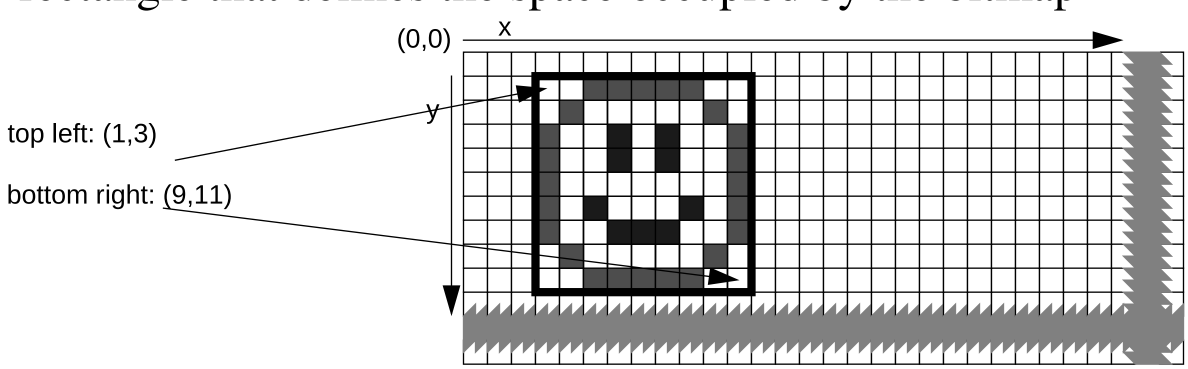
- The computer screen is organized into a rectangular grid of *pixels* that display in various colors. The coordinate system places the origin—point (0,0)—at the upper left corner and the x coordinates increase to the right, y coordinates in the down direction.



- A bitmap is a coordinate view of a group of pixels. In that group is stored an image



- When a bitmap is placed on the screen, it has a bounding rectangle that defines the space occupied by the bitmap



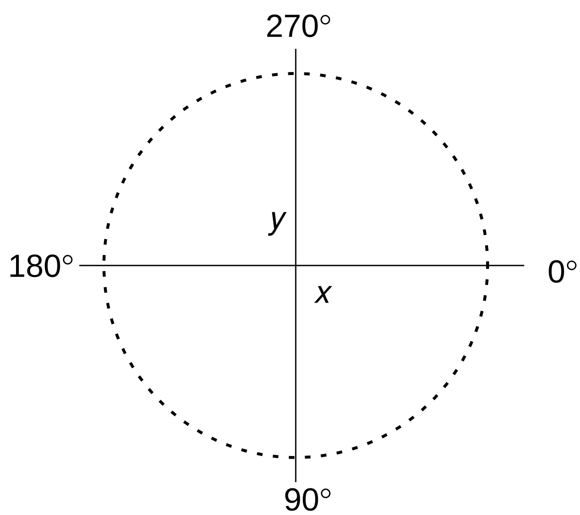
Brickles Physics

Within *Brickles*, we can model movement just as in physics—using *velocity*, which comprises a speed and a direction. If we consider a velocity to be a vector, then the magnitude (length) of the vector gives the speed, and the direction of the vector gives the direction of movement. A vector in a Cartesian coordinate system (such as that of a playing field) can be broken down into an x component and a y component. The magnitude of each component vector gives the speed in that direction.

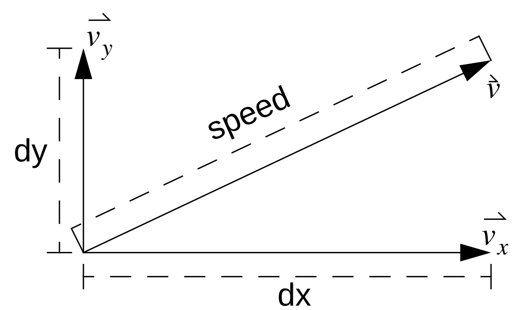
Speed is measured in "pixels per unit time interval" in *Brickles*. The unit time interval is determined by the application itself, based on the parameter used to set up the timer—for example, 20 milliseconds. You can make the puck move faster, for example, by changing its speed or by shortening the interval used by the timer. The former method is preferable for use in this course.

Brickles Physics

- Brickles makes use of "ideal" physics—for example, no friction.
- A puck moves in a playing field at any time with a given *velocity*—that is, in a given speed and direction.
  - *Speed* is measured in "pixels per unit time interval"
  - *Direction* is measured in degrees:



- *Velocity* is a vector. A vector has x and y components that give the speed (dx and dy) in the x and y directions



- Class Velocity models this concept in our design.

Exercise

Exercise

- Within Visual C++, enter the Help facility for the Microsoft Foundation Classes and locate the description of the class CRect. Make a note in the space below about how to do the following:
  - Construct a new instance of CRect
  - See if two rectangles r1 and r2 (CRect instances) intersect
  - Get the extent of a rectangle r1
  - See if one rectangle r1 lies outside—partially or completely—of another rectangle r2

Increment 1

We have decided to implement it in two *increments*:

- *Increment 1* will be our first use of MFC, including the Document-View framework, and let us become familiar with some of the library classes and the concepts they use—"bounding rectangles", timers, and so on. By the end of this increment, we would like to have a simple application that shows a puck moving around on a playing field in a window on the screen. The puck will bounce off the boundaries of the playing field in accordance with the laws of physics specified by the requirements. Increment 1 will also show a paddle moving under mouse control, but collisions with the puck and the paddle will not be supported—that is, the puck and the paddle can just move through each other. This increment will also support pausing and resuming play, and ending the game before it is over.
- *Increment 2*—will implement the rest of the requirements of the game, including collisions, the bricks, pucks absorbed by the floor, and ending the game (one way or the other).

In this unit, we focus on Increment 1.

Increment 1

- The "functionality" to be incorporated in Increment 1 is:
  - the playing field
  - the paddle moving based on mouse move events
  - a puck moving based on the timer events within the boundaries of the field.

  No collisions will be detected. However, the game can be paused (by a press of the left button on the mouse) and either resumed or ended.

- Our goals for Increment 1 are to
  - gain familiarity with the development environment,
  - understand MFC and the Document-View application framework,
  - verify that timer and mouse events can be processed fast enough, and
  - become comfortable using graphics concepts and be able to animate the puck and the paddle figures on the screen.

Note that this first increment will be based on the Document-View framework provided by MFC, which includes handling of timer and mouse events. However, we have tried to minimize the amount of knowledge you need about the Document-View framework.

Design Notes

It is difficult to capture all that goes into the design of a system. Certainly as design and implementation proceed, our understanding of the problem increases and we reflect that knowledge in updates to the analysis models. The development of what we are calling “Increment 1” required learning Visual C++, learning MFC and the Document-View framework, learning about computer graphics and animation, and learning about arcade games.

We have made every effort to hide Document-View from you. [This is not intended as a course in Windows applications development!] You really only need to understand a few basic concepts: the concepts of Document and View, and the input event that Brickles (a document) must process: timer events and mouse move events.

If you will be involved in the object-oriented design of applications, it is really in your best interest to learn about Document-View and other application frameworks because they provide a flexible, extensible architecture for many kinds of applications.

We will next look at another design for a Brickles game [that we rejected] and consider its strengths and weaknesses.

Design Notes

- Keep in mind that our goals include a set of reusable components. Eventually we would like to be able to develop a framework from which to build other arcade games. We have chosen, however, to address only *Brickles* and to postpone thinking about the more abstract concept of arcade games and frameworks because our resources are limited.
- We have tried to isolate the MFC environment from our design as much as possible in order that the course be able to focus on object-oriented design and C++ implementations instead of the development of Windows applications. However, an understanding of our use of the Document-View framework might help you to design future applications, whether they run under windows or not.
- The design presented in this course for Increment 1 has been developed iteratively—that is, we have made several passes over the analysis, design, and implementation activities. As is typical, this effort followed an incremental, iterative process. The first two iterations for Increment 1 involved:

1. Studying the design and implementation of an existing Brickles application.
2. Prototyping a small application to gain familiarity with the Visual C++ and MFC environments.

Review of an Existing Design

When doing our analysis, we tried to look at similar problems. We were able to find documentation for another Brickles implementation. The design was based on a three-level architecture:

- A “reusable layer” comprising abstract classes such as “Movable” and “Obstacle” that could apply to a variety of arcade games
- A “logical layer” that defined the concrete classes needed for Brickles and that understood the rules of the game—how a puck bounces off a wall, for example.
- A “visual layer” that comprises all the objects that are displayed on a screen—that is, that have bitmaps defined for the objects in the logical layer.

We will look next at an object model for the existing design. This design did not make use of Document-View, but did make use of classes in MFC.

Review of an Existing Design

- The design of an existing Brickles application was based on a three-level architecture:
  - a “reusable” layer that establishes abstractions of “movable” and “obstacle” and relationships between them,
  - a “logical” layer that understand the abstractions and rules for the game, and
  - a “visual” layer that comprises the objects that need to be displayed.
- The viewable layer is derived from the classes in logical layer—for example, the class Puck in the logical layer has a subclass “VPuck” that augments a puck’s functionality to be “visual.”
- An object model for this design is given on the next page.

Existing Design Object Model

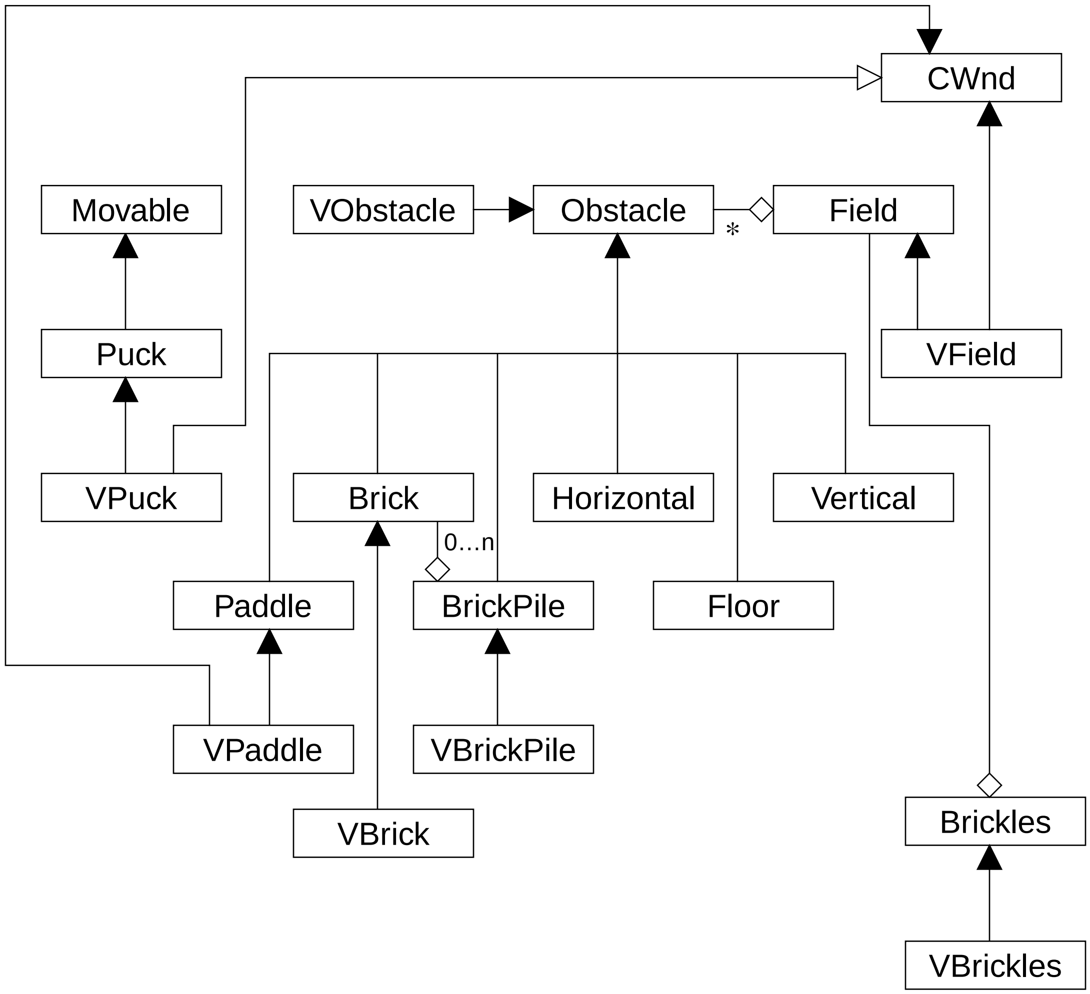
The object model shows how the layers are obtained—through generalization (inheritance). The classes in the viewable layer (named V__) are derived from classes that define concrete entities in Brickles—for example, VPaddle is the visual layer class for Paddle in the logical layer. The classes in the logical layer are derived from those in the abstract layer—for example, Brick derives from Obstacle.

We rejected this model for several reasons:

- It has poor extensibility. Consider, for example, if we were to revise Brickles to include different kinds of bricks—perhaps "tough" bricks that have to be hit twice before they break. If we make ToughBrick derive from Brick, then what do we do about VToughBrick? Make it derive from both ToughBrick and VBrick? [Why is multiple inheritance always used with great caution?]
- The implementation did not use the Document View framework, but did use some MFC classes—namely, CWnd. The movable objects inherit from CWnd, meaning that there are several windows on the screen at a time. This design displayed a curious behavior: the paddle only moved when the mouse cursor moved outside of one end or the other of the paddle window (the paddle window chased the mouse cursor). This did not provide a good "feel" of the paddle for players.

However, the design was useful because it helped us identify some useful classes—Obstacle, and BrickPile, for example.

Existing Design Object Model



- We identified some problems with this design:
  - Poor extensibility
  - “Oddities” in design and in performance
- But borrowed some ideas!

Prototyping

We identified some benefits of using the application framework—primarily that it would save us a lot of work! In order to learn more, we tackled a smaller problem first and set out to build a simple prototype. Fortunately, Microsoft provided in the documentation a program that makes a ball bounce around in a window. The ball moves using a timer, so that CPU clock speed doesn't control the ball speed. By studying this program, and changing it to incorporate a puck, we were able to figure out how to use the framework for Brickles. We were also able to discover something we hadn't expected: mouse move events are given higher priority than timer events by Windows. [We noticed this because if we kept the paddle moving, the ball barely moved!]

One piece of advice about prototypes: **throw prototypes away!** Learn what you can, salvage some things if you'll need them, but don't try to evolve the prototype code into the application. The programmers of Increment 1 did not follow that advice and a month later they were still removing remnants of code from Microsoft's ball bouncer. [You might discover a comment that mentions "ball" and not "puck."] Furthermore, we discovered later that the Microsoft's code did not use the framework facilities in the best way, so we had to back out code that we wouldn't have put there in the first place.

Prototyping

- We decided to base our application on the Document-View framework because that *should* reduce our level of effort.
- Not having used the framework before, we started with a simple example from Microsoft's documentation for Visual C++: an application in which a ball bounces within the boundaries of a window.
- We augmented the code in that example to use mouse move events to control a paddle that moved horizontally within the window boundaries. Then we added the ability for a user to pause the action via a click on the left button, and to either resume or terminate the action via a choice in a dialog box.
- This prototype showed us:
  - how to [not] use the Document-View framework,
  - a way to handle timer events,
  - that mouse move events are given priority over timer events in the framework,
  - that if events are not handled quickly enough, that memory can be exhausted
- From there, we moved on…

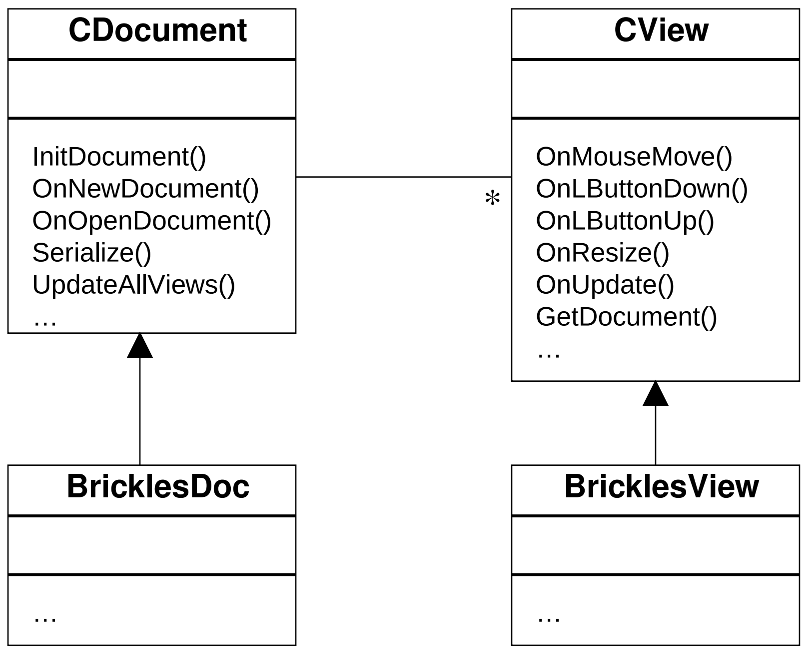
Brickles Architecture Overview

We have used the Document-View framework to define the basic architecture of our system. Recall that the view handles all input and output events, while the document handles all state and computation information. Recall, too, that an object-oriented design attempts to derive the architecture from the problem—Brickles in this case. We will build that model within the document, and include any information in the model that the view will need.

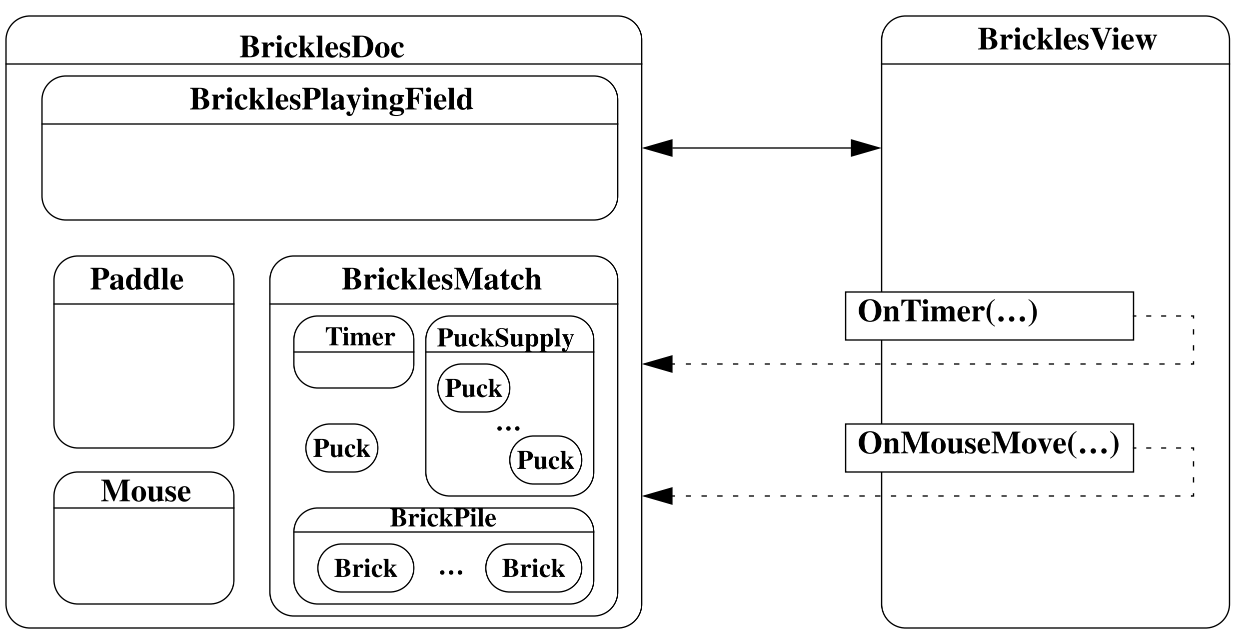
Document-View is an event-driven architecture—that is, its inputs are taken from a sequence of events that occur in the surrounding environment. In our application, we will be interested primarily in mouse move events (to control the paddle) and in timer events (to control the puck). Other events will be ignored, except for a "button down" event on the left mouse button, which will terminate the application.

Brickles Architecture Overview

- The MFC Document-View framework provides the top-level architecture for our application.



- We model the world of Brickles in the document.



WARNING!

One of the implications of using MFC is using other tools to generate an application. One such tool in the Visual C++ is the App Wizard. It annotates code files with macro calls and special comments. Do not change anything that looks like it might make sense to Visual C++, such as the fragments of code shown here.

WARNING!

- Do not modify code generated by AppWizard or ClassWizard—for example, strange-looking comments:

```
   //{{AFX_MSG(CBricksApp)
   //}}AFX_MSG
   DECLARE_MESSAGE_MAP()
```

or:

```
/*
#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif
*/
////////////////////////////////////////////////////////////////////////
// BricksView

IMPLEMENT_DYNCREATE(BricksView, CView)

BEGIN_MESSAGE_MAP(BricksView, CView)
   //{{AFX_MSG_MAP(BricksView)
   ON_WM_LBUTTONDOWN()
   ON_WM_CREATE()
   ON_WM_SIZE()
   ON_WM_MOUSEMOVE()
   ON_WM_TIMER()
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////////////////////////////////////////
```

- Visual C++ extends C++ by placing pragmas in comments and using a set of macros (#defines).

Brickles Environment

Each event in the Windows environment is routed to an application via [one of the] views. We will need to be concerned about timer events and mouse move events.

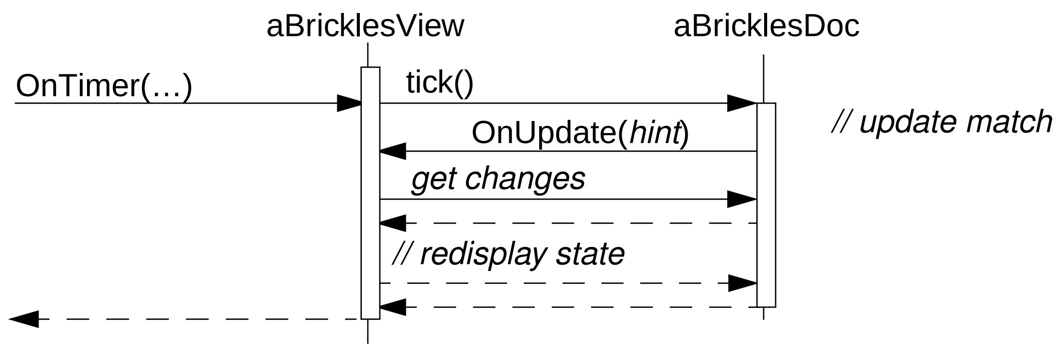
A timer event—that is, the expiration of some duration of time—is routed to a view via the OnTimer() member function call. In our design, the event is forwarded to the document via a tick() member function call. The document handles the event—in Increment 1 by moving the puck to a new position in the playing field, processing any collision with a wall, and then notifying the view of an update (via an OnUpdate() member function call). [We will see that the event is actually handled within a Brickles document by a timer object that is responsible for notifying "interested" objects that a unit time interval has passed.

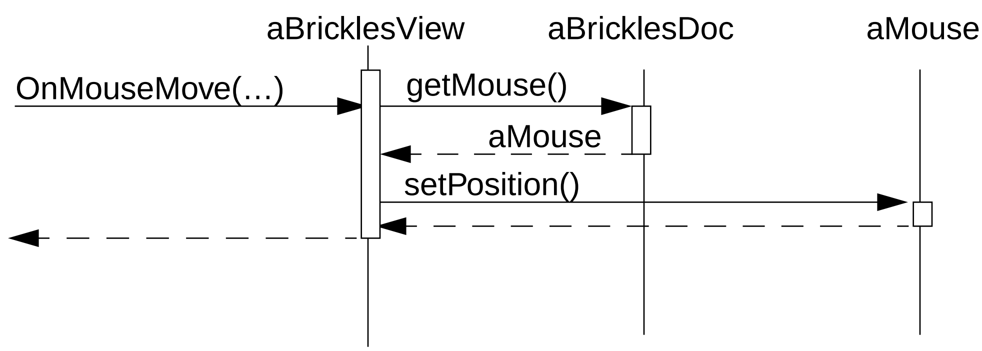
A mouse move event is handled by the view getting access to a mouse object within the document and then setting the position stored in that object to the point to which the mouse's "hot spot" has moved. Note that this design is typical of that promoted by Microsoft in its documentation of Document-View, but that this design is more in the flavor of a procedural design than an object-oriented design, which attempts to distribute responsibility. That is, instead of a view's retrieving a mouse reference from a document, we would prefer to pass the new point to the document and let it do with it what it wants to do. [This is what happens in the case of the timer.] The latter design is better because a view has more limited knowledge of how a document works. If, for example, we decide to implement the document in some other way to not use a mouse object, then we have to reimplement OnMouseMove() in the view class.

It frequently pays to review a design and implementation from time to time to identify and "fix" such things.

Brickles Environment

- The design separates responsibilities as follows: the BricklesView object interacts with the Windows environment and routes each event to the BricklesDoc object, which processes them. The view updates the screen, if necessary.





Increment 1 Design Object Model

We want to incorporate a model of Brickles into our architecture. This object model for Increment 1, which builds on the analysis object model, shows the structure of the problem we will use in Increment 1. Note that some new classes—including CView, CDocument, Velocity, CPoint, CRect, and CBitmap—have been introduced. These are computer artifacts—things we need to represent and solve the problem in the computer.

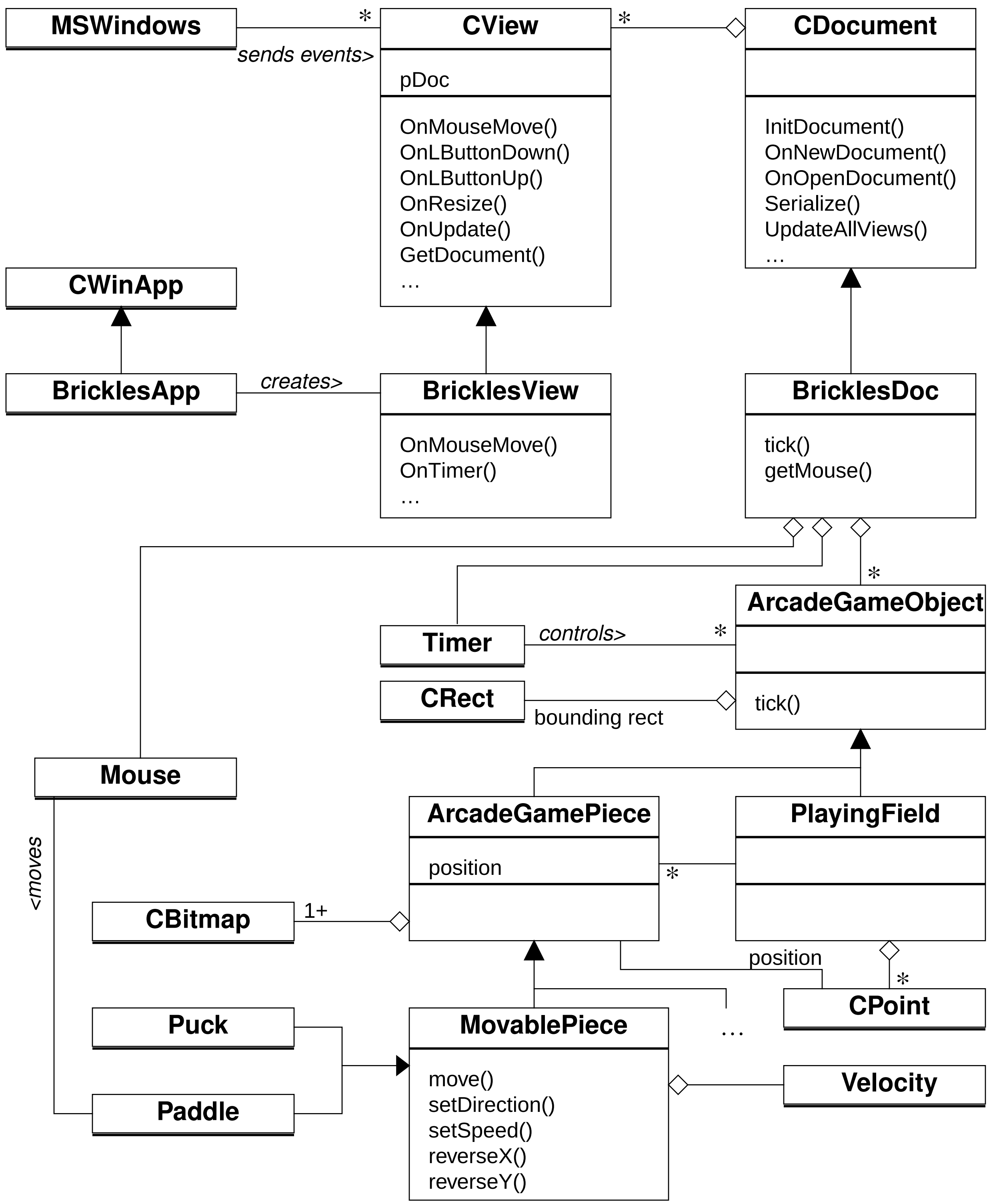
This model gives a fairly high-level perspective on Brickles. It shows the object classes and their relationships and some responsibilities:

- An *arcade game object* is an element in an arcade game. This is a generalization of *playing field* or an *arcade game piece*, one of the kinds of objects that appears in a playing field—for example, a puck or a paddle. Every arcade game object has a *bounding rectangle* and a *bitmap*.
- A *movable piece* is an arcade game piece that can move around in a playing field, such as a puck or a paddle. There are other kinds of arcade game pieces [that don't move] that we expect to find later, but that we don't need for the design of Increment 1, so we have chosen not to enumerate them.

We will see more details about this model on other *sheets*[1]. We will also see some interaction diagrams for a few different scenarios.

1. One way to organize an object model is to put different portions of it on different pages (sheets). In a set of sheets, any class C that appears on one sheet corresponds to class C on a different sheet. The associations and generalizations for class C is the union of all such relationships across all sheets. The same holds for attributes and operations.

Increment 1 Design Object Model



Increment 1 Design Object Model

We will now look at more details about the design of Increment 1:

- Class models and generalizations
- Interactions that occur for various scenarios: start-up, moving the paddle, moving the puck, stopping the game (Increment 1).

Increment 1 Design Object Model

- We will now address the descriptions of the classes in the model and then we will examine some of the basic scenarios (objects and their interactions) associated with Increment 1:
  - Starting up the application.
  - Moving the paddle within boundaries under mouse control.
  - Moving a puck within boundaries under timer control
- A class description comprises descriptions of
  - the concept or entity the class represents,
  - the observable states of an instance (if more than one exists) and the transitions between them,
  - the [public] protocol of an instance, and
  - any other supplemental information.
- We will show descriptions for some of the classes that we have identified as being needed for Increment 1. For the sake of simplicity, some details associated with Windows programming will be omitted from these descriptions.
- NOTE: All member function bindings should be interpreted as *virtual* in the descriptions that follow unless noted otherwise.

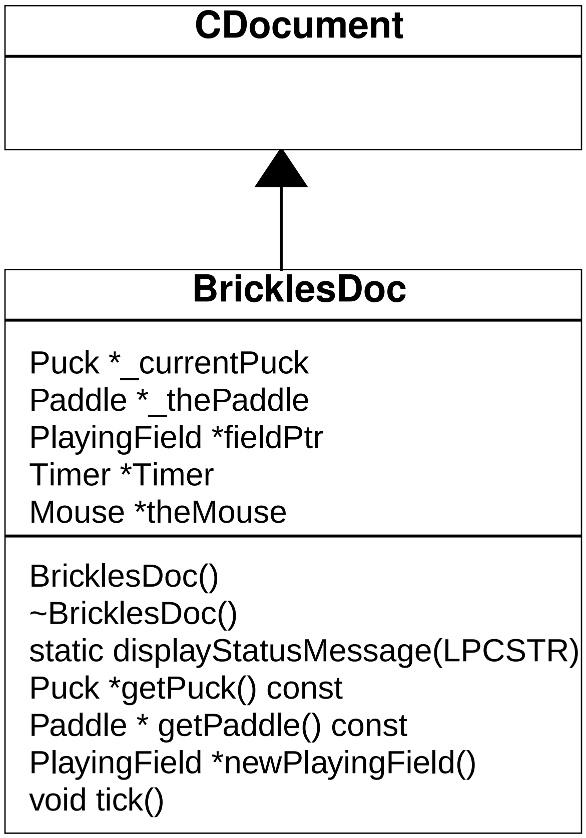
CDocument

Most of what BricklesDoc needs to be a document is inherited from the MFC CDocument class. We have to add features that supports our game model. For increment 1, those include a paddle (and a mouse object needed to manipulate it), a puck (and a timer needed to move it), and a playing field.

The accessor operations are self-explanatory.

The tick() operation is used to inform a Brickles document that a timer event has occurred.

CDocument



BricklesDoc is an arcade game played with a puck and a paddle (see the specification).

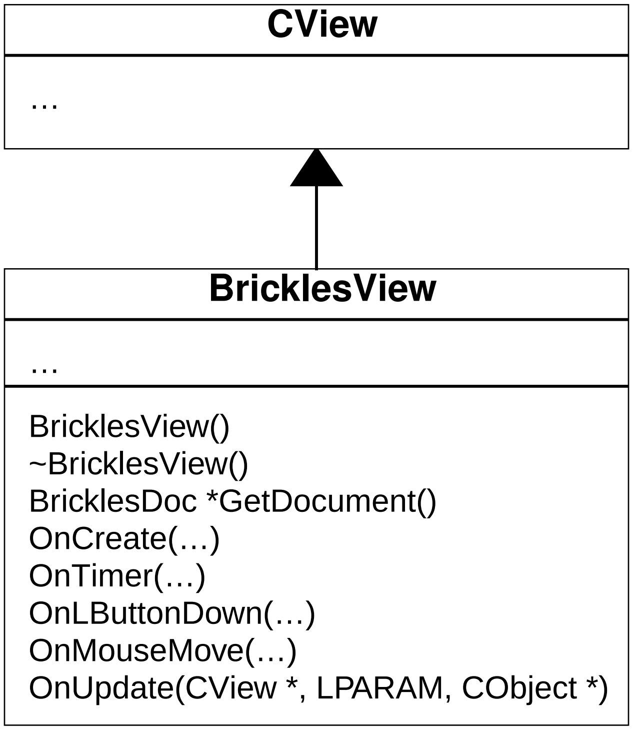
NOTES:

- The static member function BricklesDoc::displayStatusMessage() can be used to display a short string in the lower left frame of the Brickles window—example: BricklesDoc::displayStatusMessage(“Hello!”)

CView

The view for a Brickles document displays the state of the game at any particular time. In Increment 1, the state is reflected simply by the positions of the paddle and the puck in the playing field.

CView



A Brickles view displays the state of a Brickles game at any particular time.

NOTES:

- You won't have to be concerned with most of the operations associated with a BricklesView. You will see some of these appear in member function call stacks within the debugger, however.

ArcadeGameObject

Arcade game objects comprise those elements of Brickles that are used to play the game, including a playing field, a puck, a paddle, and (in later increments) bricks. Every arcade game object has a bitmap and a bounding box. [Design note: in its implementation, the Brickles playing field does not use a bitmap; it relies on the background of the containing window for its implementation.

ArcadeGameObject

ArcadeGameObject

CRect _boundingRect
CBitmap *objBitmap

ArcadeGameObject(CBitmap *) const
~ArcadeGameObject()

CRect getBoundingRect() const;
CBitmap* getBitmap()
CBitmap* getBitmap() const
void tick()=0

*An **arcade game object** is any component of an arcade game, including the playing field and any other pieces. Since an arcade game object is visible at least sometimes, it has associated with it a bitmap that portrays its current state.*

PlayingField

PlayingField()
PlayingField(PlayingField &)
~PlayingField()
void tick()

ArcadeGamePiece

CPoint _posField
PlayingField *_fieldPtr

ArcadeGamePiece(PlayingField*,
CPoint &)
ArcadeGamePiece(
ArcadeGamePiece &)
~ArcadeGamePiece()
CPoint getPosition()
PlayingField *getPlayingField()

*A **playing field** is a two-dimensional area in which Brickles is played. The area comprises a finite set of discrete locations, addressed using Cartesian coordinates. A playing field is rectangular in shape, bounded by four walls, and contains a brick pile, a paddle, and a puck.*

*An **arcade game piece** is a component of an arcade game that can appear on a playing field. As such, every arcade game piece has a position within the field and a "footprint" within the field. Associated with that footprint is a bounding rectangle.*

NOTES:

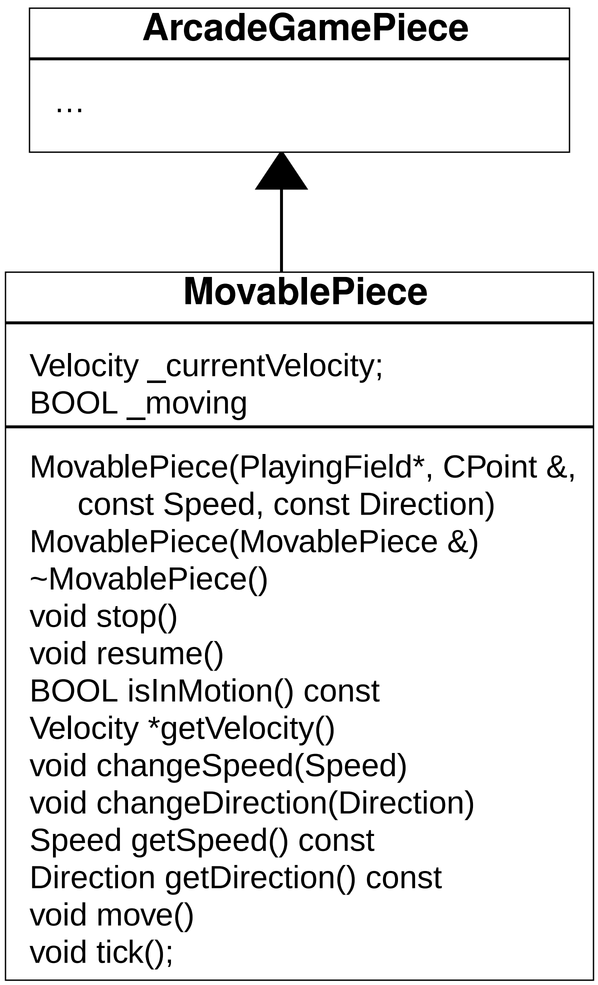
- The association between a playing field and an arcade game piece is implemented only in one direction—from ArcadeGamePiece to PlayingField (using _fieldPtr).

Movable Piece

A movable piece is an arcade game piece that moves around in a playing field.

A dynamic model for a movable piece is also shown here.

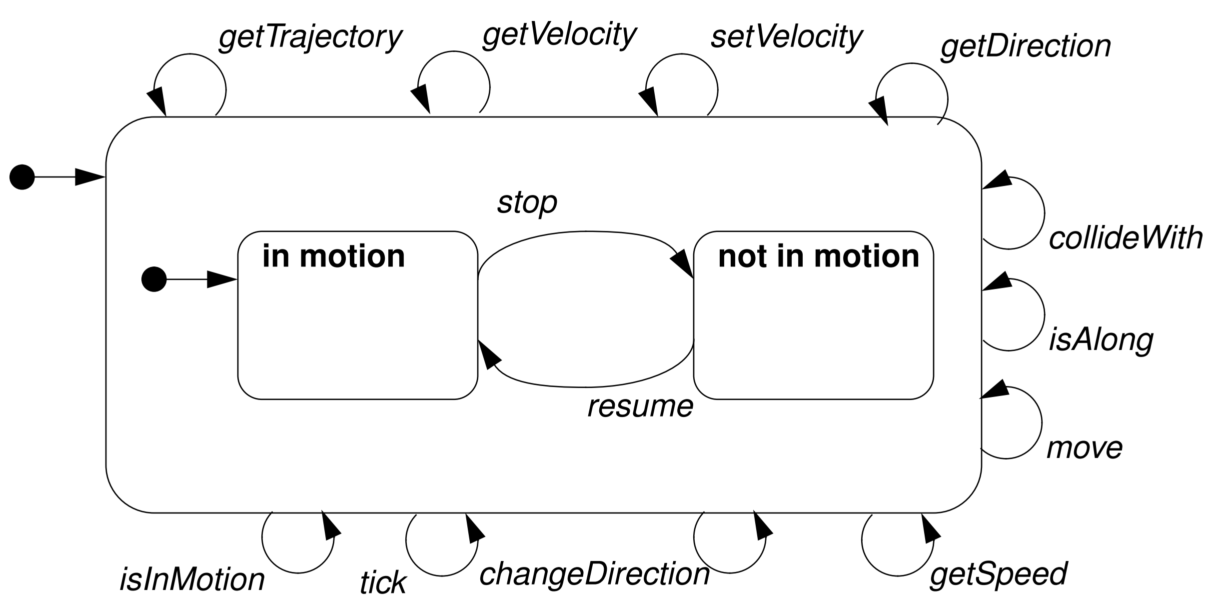
MovablePiece



*A **movable piece** is an arcade game piece that can move in a playing field during a match. Each movable piece has a velocity which is used to determine its next position in the field*

The position of a movable piece in a playing field is the center of its bounding rectangle.

A movable piece is not always necessarily in motion. If it is in motion, then it moves when told to do so. Otherwise a request to move has no effect.

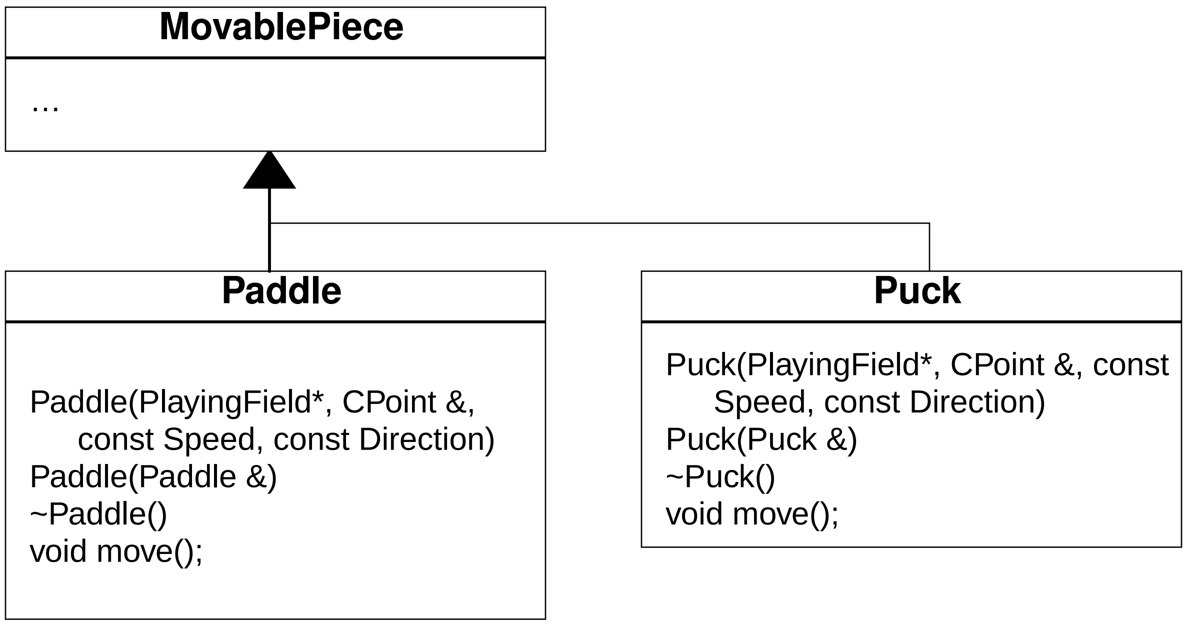


Puck and Paddle

Both pucks and paddles can move in a playing field, though under different controls. A paddle moves under player control, while a puck moves autonomously—that is, based purely on the passing of time.

These classes define the difference between the way these objects are controlled. It is interesting to consider whether the idea of a velocity is meaningful for a paddle. In some sense, the paddle jumps around depending on the sampling rate of mouse moves. The puck moves smoothly—as smoothly as is possible in discrete time! A paddle is constrained in the way it can move, too: the y component of the speed must be zero. Is there truly an “is a” relationship between Paddle and MovablePiece?

Paddle and Puck



*A **paddle** is a movable object that is controlled by a mouse.*

*A **puck** is a movable object that moves autonomously—that is, it is not under user control, but advances to a new position in the playing field according to the laws of physics*

Velocity

Velocity represents the concept of a vector as described earlier.

It is interesting to note that both _direction and its components, _dx and _dy, are stored in a velocity. The information is redundant since the constraint: (_direction * _direction) = (_dx * _dx) + (_dy * _dy) must hold. However, since integers are being stored, round-off error might result in this constraint not always being satisfied. Thus, we maintain both sets of values. Movement through a playing field is done “approximately” using the components, but a velocity object will always reply its “true” speed when asked.

Velocity, as a concept, could be developed quite a bit further. We could add in operations to support direction given in radians, for example, in addition to the other operations listed. However, because we think this is sufficient development for our needs in Brickles—and probably most other kinds of arcade games we might want to implement—we are content to go no further.

Velocity

```
typedef int Speed;
typedef int Direction;
```

Velocity
Speed _dx Speed _dy Speed _speed Direction _direction
Velocity() Velocity(const Speed, const Direction) ~Velocity() Speed getSpeed() const Speed getSpeedX() const Speed getSpeedY() const void setSpeed(const Speed) Direction getDirection() const void setDirection(const Direction) void reverseX() void reverseY()

*A **velocity** is a rate of change in position made by an object over a unit interval of time, characterized by a speed and a direction in some Cartesian coordinate system. A velocity is a vector, having a direction and magnitude in the coordinate system.*

NOTES:

- Velocity uses two other types: Speed and Direction
- Velocity could be defined to encompass many more concepts associated with vectors—adding two vectors together to get another, computing projections on various bases, and so on. We have chosen to restrict it to our needs for Brickles, ignoring the opportunity of reusing this class in other contexts.

Timer

We have incorporated a timer object to represent the hardware device on the computer which is made available to us via the Document-View application framework. It is quite common in object-oriented design to define an object to represent a hardware component of a system.

The purpose of our timer is to distribute the occurrence of the passing of a unit interval of time to any interested objects in the game. Any interested object can be registered with the timer, via an enroll() member function, and the timer will notify such an enrollee by sending it a tick() message.

Note that we allow any instance of any subclass of ArcadeGameObject to enroll with a timer, even though it might seem more accurate to allow only "movable" pieces to register. However, we have thought ahead to possible enhancements that might allow time-based state changes within non-movable pieces—for example, bricks might change their colors after every 100 ticks, or floor might reflect pucks for a while after some condition has occurred during the play of the game. Note that to support this capability, we have put "tick()" into the protocol of ArcadeGamePiece to support the timer operation as we have described, and we have provided ArcadeGameObject::tick() with a "no-op" implementation to provide a reasonable default behavior for this operation in subclasses.

Timer

Timer

int _enrolleeCount
ArcadeGameObject *_enrollees[MAX]
BOOL _enabled

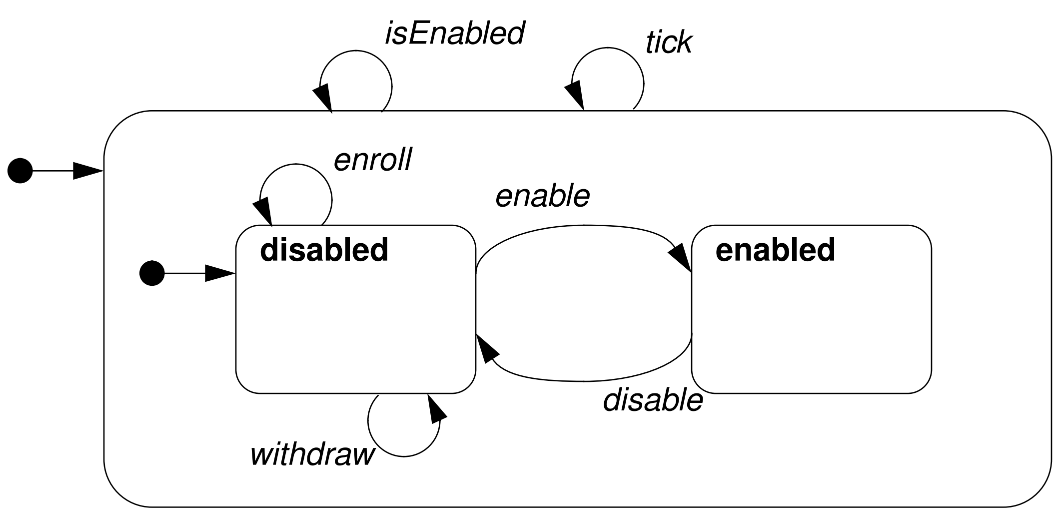
Timer(int max, int unitInterval)
Timer(Timer &)
~Timer()

void enroll(ArcadeGameObject *) const
void withdraw(ArcadeGameObject *) const
BOOL isEnabled() const

void enable()
void disable()

void tick() const

*A **timer** is a device that alerts up to some maximum number of arcade game objects as to the passage of a unit interval of time. A arcade game object can be enrolled with a timer. When a unit interval passes, each enrolled object is sent a "tick()" message. A arcade game object can be withdrawn from the enrollment list for a timer. A timer can be disabled to inhibit notification of all enrollees.*



Mouse

The hardware mouse, like the timer, is represented in our design by an object. We have a very limited model for a mouse device, however. It is restricted to modeling the current mouse location. It does nothing with mouse button positions, nor does it allow us to move the mouse cursor to another location on the screen. It does just what we need to do for Brickles during play—exactly what the Brickles document is modeling.

We identified the need for this class in doing our prototype. We discovered from that exploration that Windows gives higher priority to mouse move events than to timer events. If, in playing with our prototype, we moved the mouse constantly, then the paddle would move, but not the puck. It also appeared that if we moved the mouse long enough, we would run out of memory as timer events were buffered.

Our solution to this was to try to get mouse events processed as quickly as possible. [In the prototype, a mouse move caused the paddle bitmap to be erased and then redrawn in a new position.] To that end, we decided to process a mouse move by, in essence, storing the new location and then retrieving it whenever there was a timer tick. We figured that the sampling rate for the mouse had to be greater than a unit time interval on the timer. Thus, we made our paddle object time-driven. When a paddle gets a “tick()”, it reads the mouse to get a reading of the latest mouse position. If the new reading indicates a move has occurred, the paddle adjusts its position. Thus, we have converted an interrupting device to a polled device by using a mouse object in our design.

Mouse

Mouse
CPoint _currentPosition
Mouse() ~Mouse() CPoint getPosition() const void setPosition(const CPoint &);

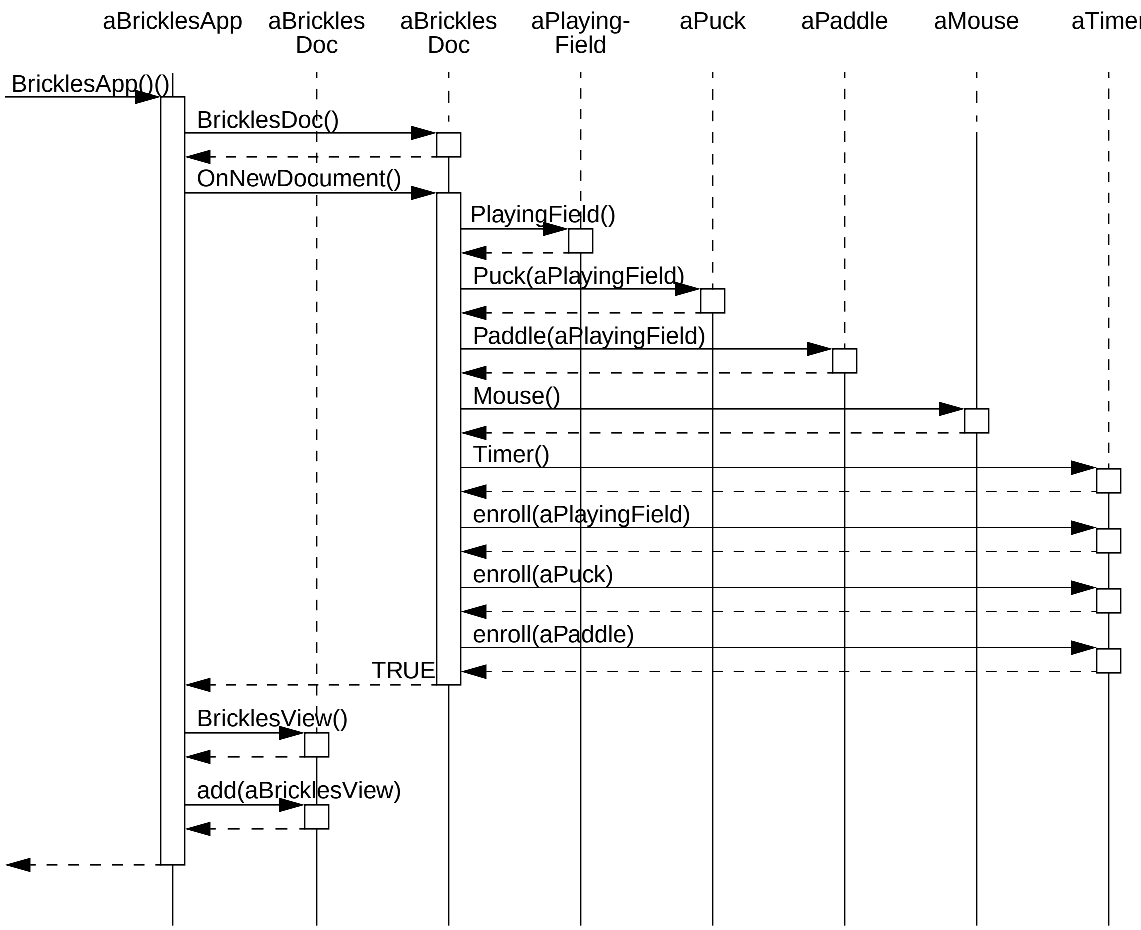
*A **mouse** is an input device used to control various movable pieces in a playing field during a match. The mouse cursor is associated with a position on the playing field when a game is in progress.*

Why doesn't a mouse work like a timer?

Start-Up Scenario

This interaction diagram shows how Increment 1 starts up—how various objects are created and related.The application is created when the program loads.

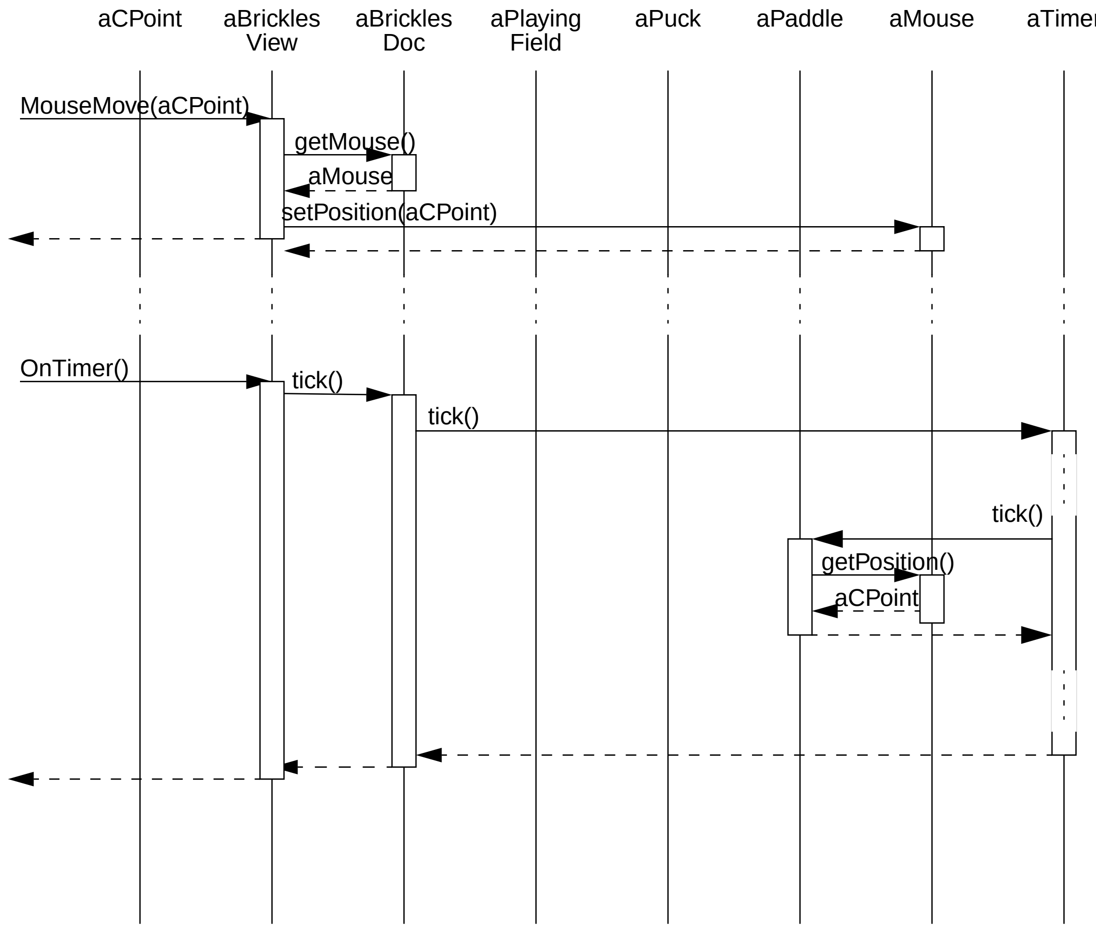
Start-Up Scenario



"Paddle Move" Scenario

This interaction diagram shows how the paddle moves in response to a player moving the mouse. As the diagram shows, the move occurs across [at least] two events: a mouse move event (OnMouseMove()) and a timer event (OnTimer()). This scenario assumes that the mouse position is changed between timer events.

"Paddle Move" Scenario

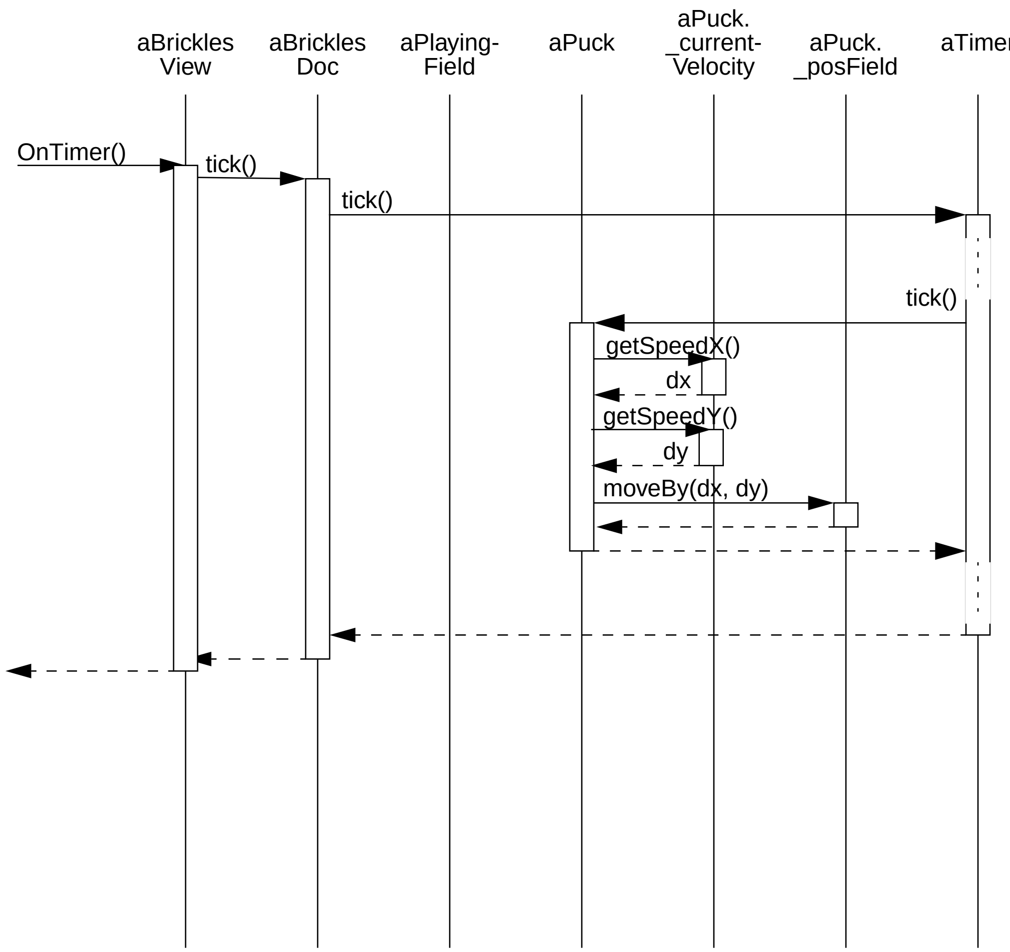


"Puck Move" Scenario

The puck moves when another unit time interval has expired, as measured by the system timer and propagated as a timer event to a view. The puck moves by computing its next position in a playing field using the magnitude of the x and y components of its velocity.

This scenario assumes the resulting position is within the playing field boundaries.

“Puck Move” Scenario



Where did this design come from?

The architecture of our application derived primarily from the MFC Document-View framework. We decided to use that in order to reduce the amount of effort required in getting a Windows application developed. We liked the advantages of using that architecture—namely, that separation of computation from input/output activities reduces the amount of rework that has to go on if we want to change the "look" of an application, but not the computational aspects.

Within the document, we have designed the structure to reflect the structure of the problem, as defined in an application object model. We have designed and implemented the links defined by the associations in the model. Some of the links were implemented in only one direction of the association, others were implemented in both directions. We also introduced some new classes and relationships. We devised a design object model that refines the analysis object model to reflect the new classes and relationships.

In order to effect the necessary computations in the system, we used interaction diagrams to figure out how an input can be transformed into an output. These diagrams helped us figure out what objects should be involved in each scenario, and what each of those objects should do. In creating one diagram, we might have identified changes that would apply to others we had already done.

Certainly other architectures exist. [We looked at one briefly at the beginning of this unit.] We could, for example, have used multiprocessing—letting the puck and the paddle and the playing field all be separate processes, just as they seem to be in the "real" world. Since the version of Windows we are using does not support multitasking, that was not an option. If we were using Java, Smalltalk or Concurrent C++ as our implementation language, then we might have a different architecture. Implementation language definitely can have an influence on design.

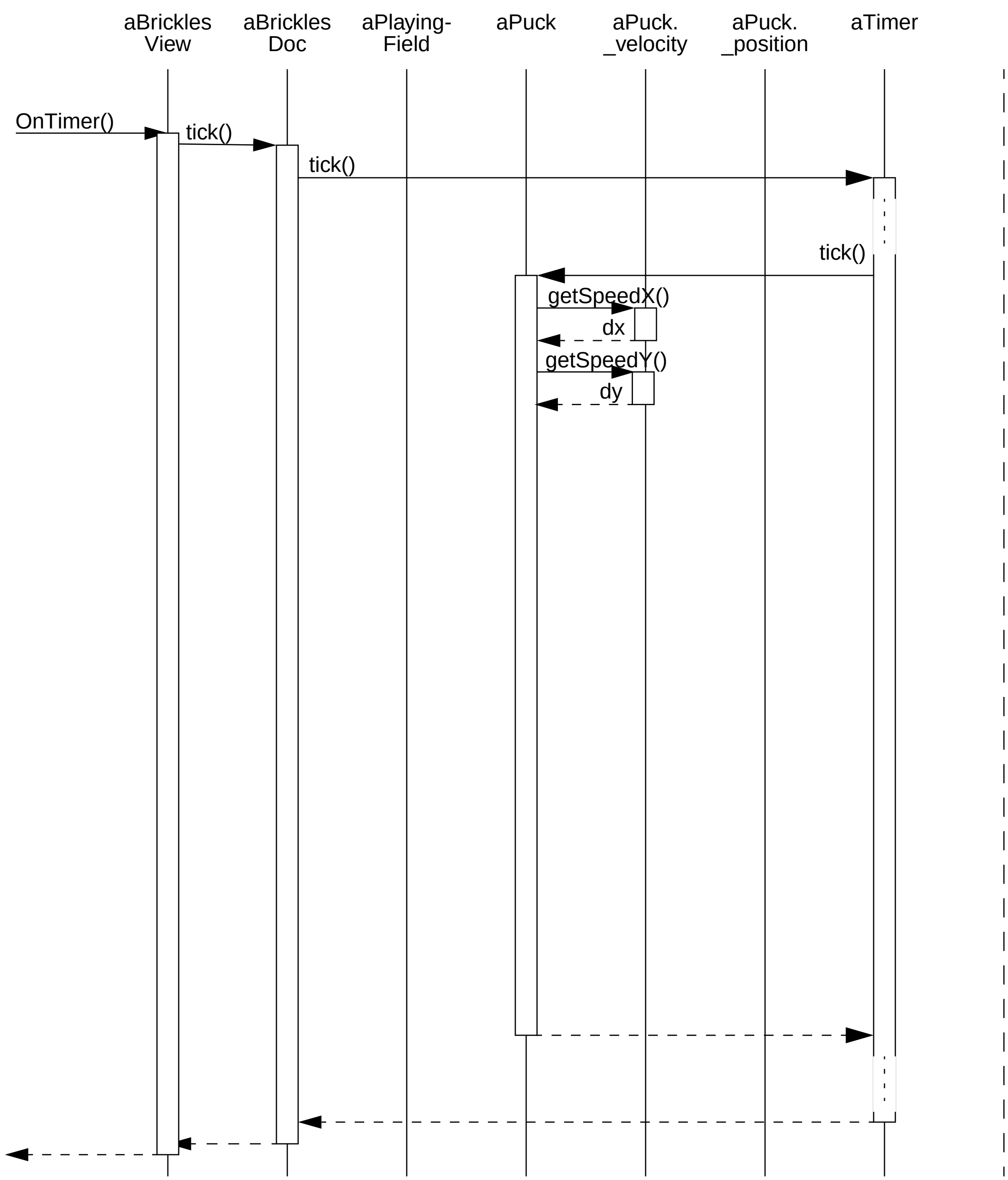
Where did this design come from?

- The Document-View framework provided the basic skeleton and division of responsibilities:
  - We created a subclass of CDocument to maintain information about the state of the game.
  - We created a subclass of CView to perform all processing associated with input and output.
- Within a BricklesDoc instance, we have used the structure of the underlying problem to provide the structure. For Increment 1, we have a playing field that contains a puck and a paddle, a puck that moves, and a paddle that moves. [The behaviors so far are taken from the problem.] The relationships between those objects are taken from the analysis object model. In that model, for example, a playing field contains arcade game pieces, of which pucks and paddles are kinds. Each ArcadeGamePiece instance maintains a pointer to the playing field containing it. The playing field, in Increment 1 anyway, keeps a pointer to the paddle and the current puck.
- We worked through various scenarios—start-up, paddle move, puck move—to see how we might get the desired behavior from these objects. We added additional objects—a timer and a mouse—to help us get that behavior.
- Why didn't we use a multitasking architecture—for example, make puck a separate process?

Exercise

Feel free to ask the instructor any questions you have about the assignment or about the design of Increment 1.

Use this interaction diagram to help you in designing a solution. *At the point dy is returned to the puck, a move by dx and dy will put the puck partly or fully outside the bounds of the playing field in which it resides.*



Exercise

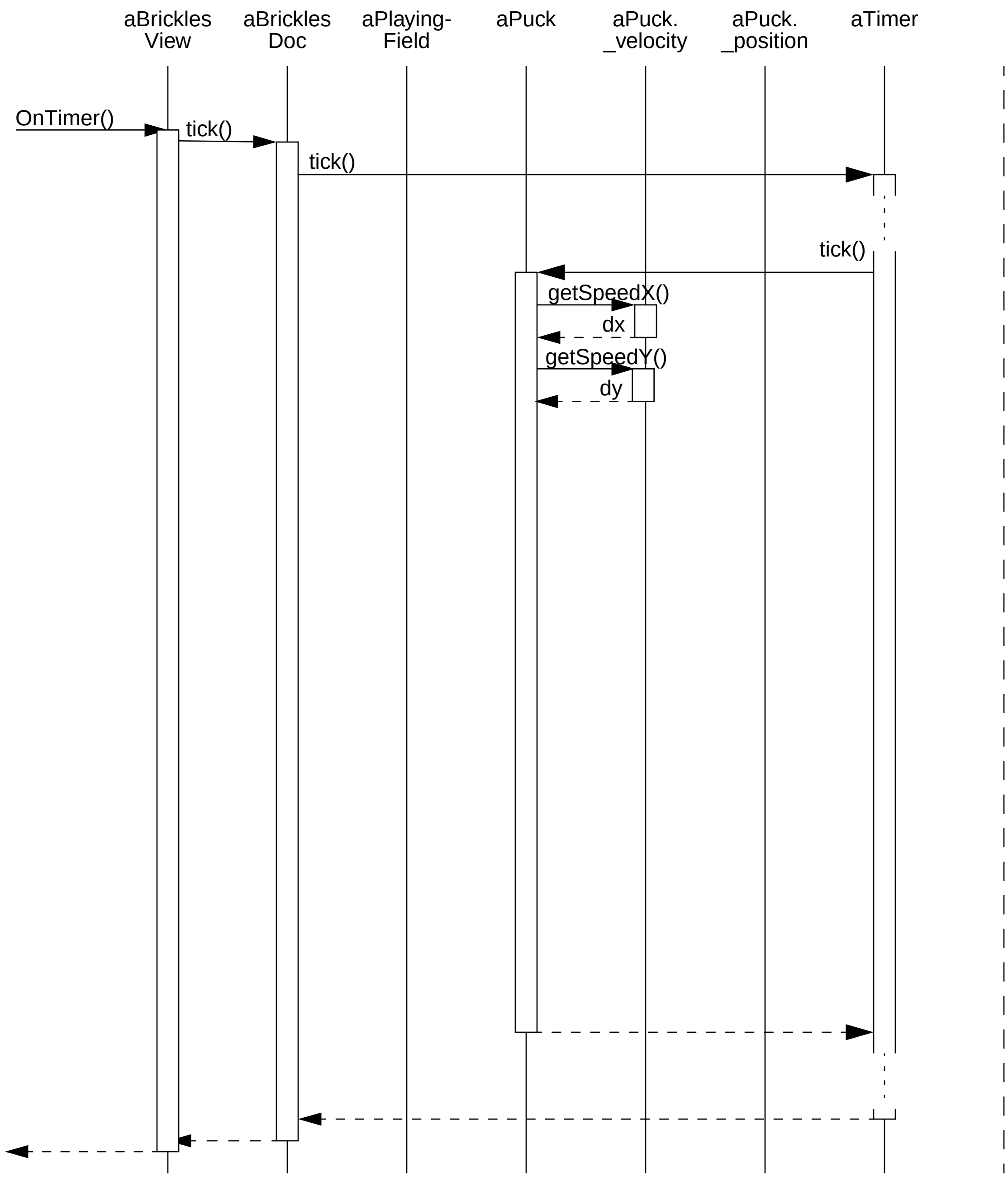
Complete Increment 1. The primary goals for this exercise are to help you get familiar with the Visual C++ environment and with the design as it has been presented so far.

- The files in directory WORKSHOP\INC1 comprise a project for Increment 1 of Brickles. The code in this project does everything we need in Increment 1 except make the puck bounce within the boundaries of the playing field. Your assignment: Design and implement code to meet that requirement.
- The project is structured such that for each class C there is one header file (C.h) and one source file (C.cpp). Since Windows and (DOS) support only 12-character file names, many of the class names have been abbreviated to yield a file name. Listings are provided in Appendix D.
- Clicking on the left mouse button will terminate execution. Recall that BricklesDoc::displayStatusMessage(LPCSTR) is available for short messages. Use the debugger!
- To complete this assignment, you can start by looking at Puck::move(). You might want to set a breakpoint there and run the application to see how you get there.

  In designing your solution, consider several solutions. Consider for example if the puck is the object that limits its own travels, or whether there is some other object that might be able to do it. The facing page contains an interaction diagram that begins a scenario you are addressing.

Optional Exercise

If you have time after completing the previous exercise, consider the problem of recognizing collisions between the paddle and the puck and the problem of handling the collision. Design and implement a solution. The scenario starts out as shown below.



Optional Exercise

- If you have time, consider that in the next increment we will want to recognize collisions between a puck and the paddle. Design and implement code to recognize these collisions and to redirect the puck as required. (Recall the specifications about collision with the near third, middle third, and far thirds of the paddle.)
- Consider the problem of the puck hitting the floor and being lost, to be replaced by another puck. How will you make that happen?

Exercise

It is sometimes useful to see what others have come up with for a design.

Exercise

- Share your design and any interesting things you noticed during implementation with the class.

Summary

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

-
-
-

Summary

- We have introduced pixels, bounding rectangles, bitmaps, and velocity needed for Brickles.
- We reviewed an existing Brickles design and found some problems
- We established goals for Increment 1 and described the design. You have had a chance to complete Increment 1, and in so doing to gain some familiarity with Visual C++, with some classes in MFC, and consider some alternatives for completing that increment.