Adele Goldberg and Smalltalk

- When object-oriented programming appeared in Europe, it didn’t attract much attention in the United States.
- The success of the object-oriented paradigm on this side of the Atlantic is largely due to the work of a group of researchers at Xerox PARC, who used it as the basis for a language called Smalltalk.
- One of the project leads was Adele Goldberg, who was the person that introduced (reluctantly, as you saw in the video) Steve Jobs to the ideas of object-oriented programming.

Extending the GObject Hierarchy

- Although designing new class hierarchies will prove useful as you move on to larger applications, it is often more useful to extend existing ones to add new behavior.
- Two classes in the GObject hierarchy turn out to be an ideal base for extension:
  - The GPolygon class makes it easy to define new GObject subclasses that have a polygonal outline. These new subclasses automatically support filling because GPolygon does.
  - The GCompound class serves as a basis for new GObject subclasses that are composed of a combination of other shapes.
- In either of these cases, it is often useful to define additional methods in the subclass to support operations that are specific to that type.

Defining a GStar Subclass

```python
def DrawOutlinedGoldStar():
    gw = GWindow(400, 400)
    gw.setFilled(False)
    gw.setPenColor("Gold")
    gw.addPoint(30, 250, "circle")
    gw.addPoint(270, 250, "circle")
    gw.addPoint(150, 0, "circle")
    gw.addPoint(150, 500, "circle")
    gw.addPoint(300, 0, "circle")
    gw.addPoint(300, 500, "circle")
```

Code for the GTextBox Class

```python
def DrawTextBox():
    gw = GWindow(400, 400)
    gw.setPenColor("Black")
    gw.setPenSize(5)
    gw.setZoom(2)
    gw.drawString("Hello World!", 100, 200)
```

Code for the GStar Subclass

```python
def DrawFlexedGoldStar():
    gw = GWindow(400, 400)
    gw.setPenColor("Gold")
    gw.addPoint(30, 250, "circle")
    gw.addPoint(270, 250, "circle")
    gw.addPoint(150, 0, "circle")
    gw.addPoint(150, 500, "circle")
    gw.addPoint(300, 0, "circle")
    gw.addPoint(300, 500, "circle")
```
Graphical Object Decomposition

- The most important advantage of using the `GCompound` class is that doing so makes it possible to apply the strategy of decomposition in the domain of graphical objects. Just as you use stepwise refinement to break a problem down into smaller and smaller pieces, you can use it to decompose a graphical display into successively simpler pieces.
- The text illustrates this technique by returning to the example of train cars from the lecture on September 13, where the goal was to produce the picture at the bottom of this slide. In that lecture, decomposition led to a hierarchy of methods. The goal now is to produce a hierarchy of classes.

TrainCar Hierarchy

- The critical insight in designing an object-oriented solution to the train problem is that the cars form a hierarchy in which the individual classes `Engine`, `Boxcar`, and `Caboose` are all subclasses of a more general class called `TrainCar`.

Nesting Compound Objects

- Given that a `GCompound` is also a `GObject`, you can add a `GCompound` to another `GCompound`.
- The `Train` class on a subsequent slide shows how to define a train as a `GCompound` to which you append new cars:

```python
train = Train()
train.append(Engine())
train.append(Boxcar("Green"))
train.append(Caboose())
```

- One advantage of making the train a single object is that you can then animate the train as a whole.

DrawTrain Program

```python
class Train(GCompound):
    def _init_(self):
        GCompound._init_(self)
        self.append(Engine(color="red"), 0)
        self.append(Boxcar(), 20)
        self.append(Caboose(), 40)

class Boxcar(GObject):
    def _init_(self, x, y, w, h):
        GObject._init_(self, x, y, w, h)
        self.append(Rectangle())
        self.append(Circle())

class Caboose(GObject):
    def _init_(self, x, y, w, h):
        GObject._init_(self, x, y, w, h)
        self.append(Rectangle())
        self.append(Circle())
```

```python
# Constants
WINDOW_WIDTH = 500;
WINDOW_HEIGHT = 300;
CAR_WIDTH = 113;
CAR_HEIGHT = 54;
CAR_BASELINE = 15;
CONNECTOR = 6;
WHEEL_RADIUS = 12;
WHEEL_OFFSET = 24;
CAR_WIDTH = 53;
CAR_HEIGHT = 12;
SKEWSTACK_HEIGHT = 12;
SKEWSTACK_OFFSET = 12;
DOOR_WIDTH = 27;
DOOR_HEIGHT = 8;
CUPOLA_WIDTH = 53;
CUPOLA_HEIGHT = 12;
TIME_STEP = 20;
```

```python
# DrawTrain.py
class DrawTrain:
    def _init_(self):
        Train._init_()
        self.append(Engine(), 0)
        self.append(Boxcar("Green"), 20)
        self.append(Caboose(), 40)
        self.append(Train(), 200)
```
The DrawTrain Program

```python
def DrawTrain():
    def clickAction(e):
        nonlocal timer
        timer = gw.setInterval(step, TIME_STEP)
    def step():
        train.move(-1, 0)
        gw = GWWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
        timer = None
        train = Train()
        train.append(Engine())
        train.append(Boxcar("Green"))
        train.append(Caboose())
        x = (gw.getWidth() - train.getWidth()) / 2
        y = gw.getHeight()
        gw.addEventListener("click", clickAction)
```

Exercise: The TrainCar Hierarchy

- As it appears in the text, DrawTrain.py does not implement the full TrainCar hierarchy. It defines the BoxCar subclass but leaves Caboose and Engine as exercises. The source file, however, does include the constants that define the geometric parameters for these types.
- For the rest of class, I will add new code for these classes as you tell me what to write. We’ll then debug the code together.