Object Oriented Design and Analysis
CPE 372

Lecture 10

Design Patterns 2

Dr. Sally E. Goldin
Department of Computer Engineering
King Mongkut's University of Technology Thonburi
Bangkok, Thailand

Using Design Patterns

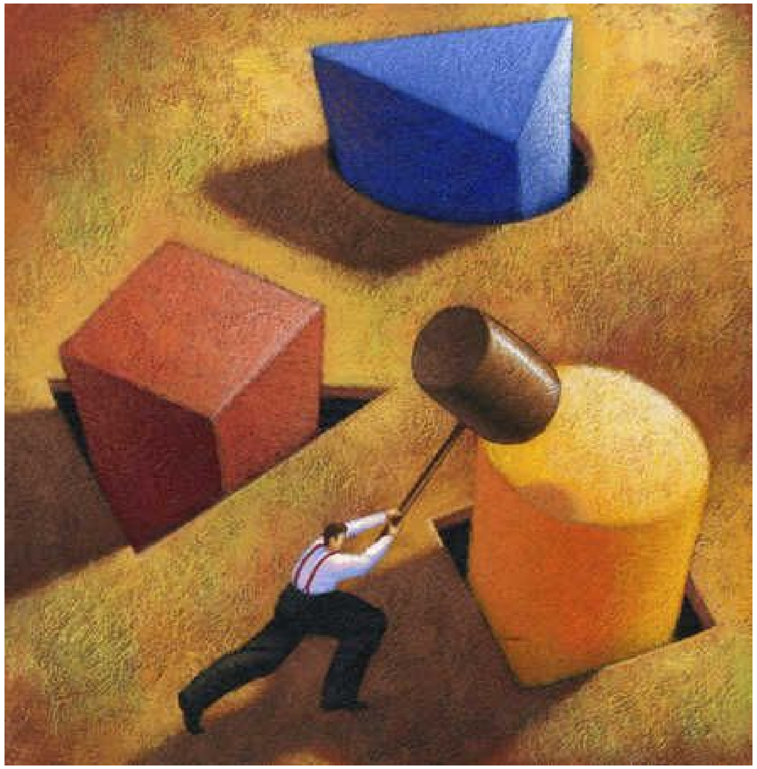
- Design patterns —quick and concise way to describe *common solutions to frequently occurring problems*
- *Accelerate the design process* by providing a ready-made outline or schema for components and their relationships
- Allow developers to communicate about their designs using a *shared vocabulary*
- Summarize and *communicate programming expertise* for less experienced developers

Problems with Design Patterns

No official, standardized set of patterns

Patterns are not ***orthogonal* –** related in complicated ways

Efforts to force software to fit a pattern can result in overly complex designs



Today

- ***Factory pattern***
- ***Observer pattern***
- ***Adapter pattern***
- ***Model-View-Controller***

The "Gang of Four" responsible for introducing software patterns: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (1994)

Factory Method Pattern

Problem

We need a general class to create instances of more specific classes, without knowing the details of the specific classes. Hence we do not want the general class to have to call specific constructors.

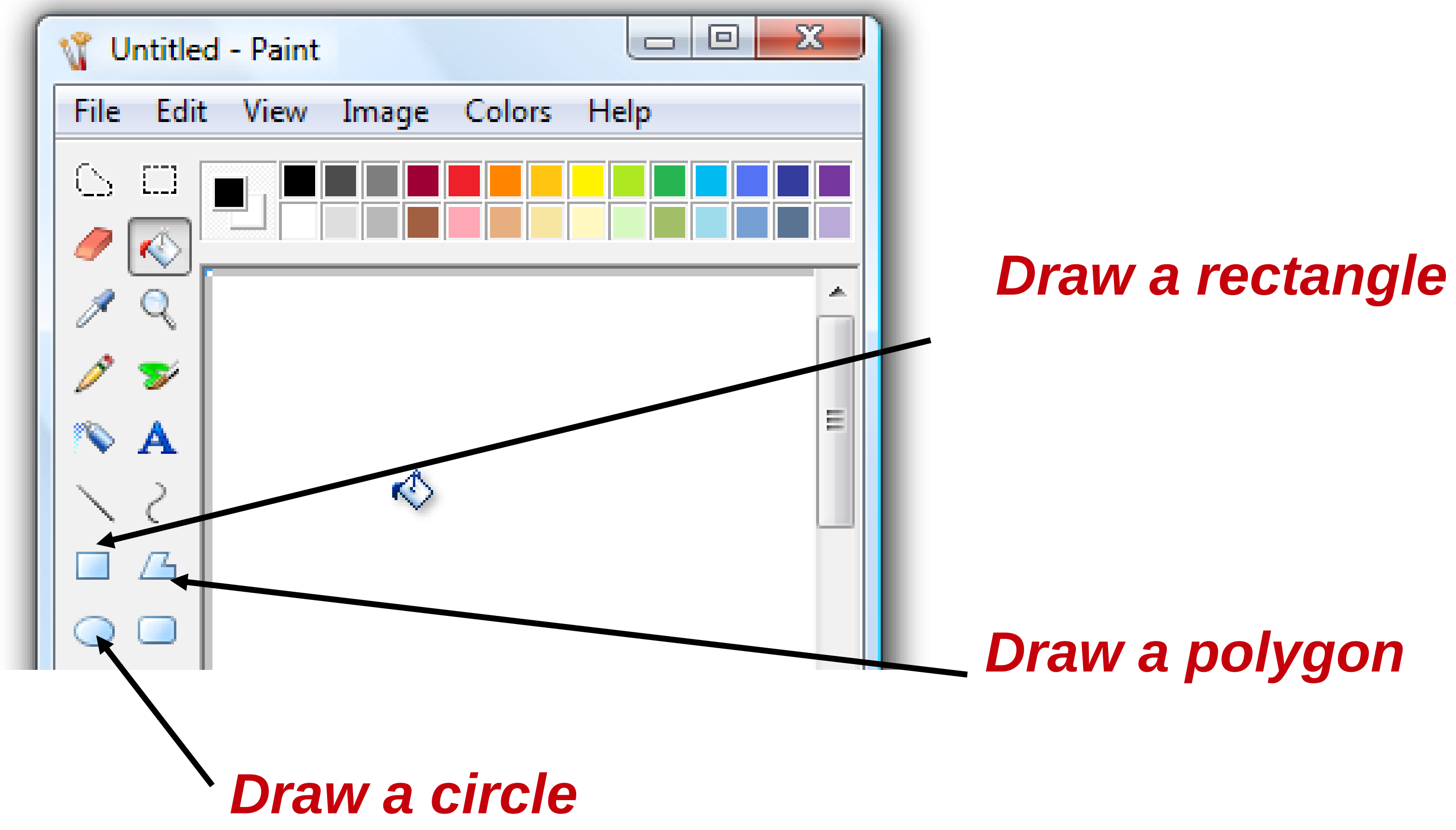
Solution

Create a super-class to hold the factory method (e.g. createInstance()), then create subclasses for each specific class.

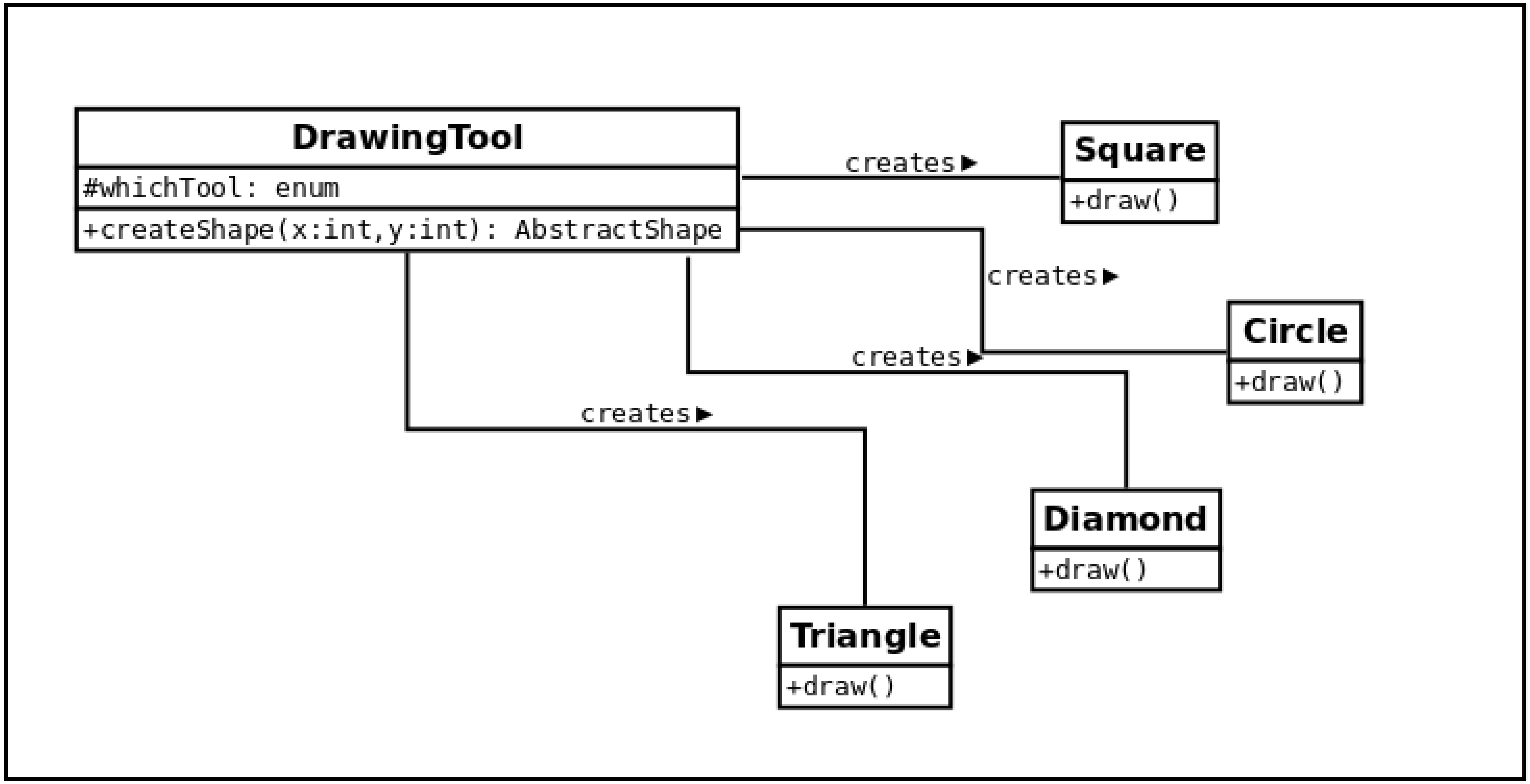
Example

Suppose we decide to use our **AbstractShapes** to create a new interactive drawing application...

Use different tools to draw different objects



Initial Class Diagram

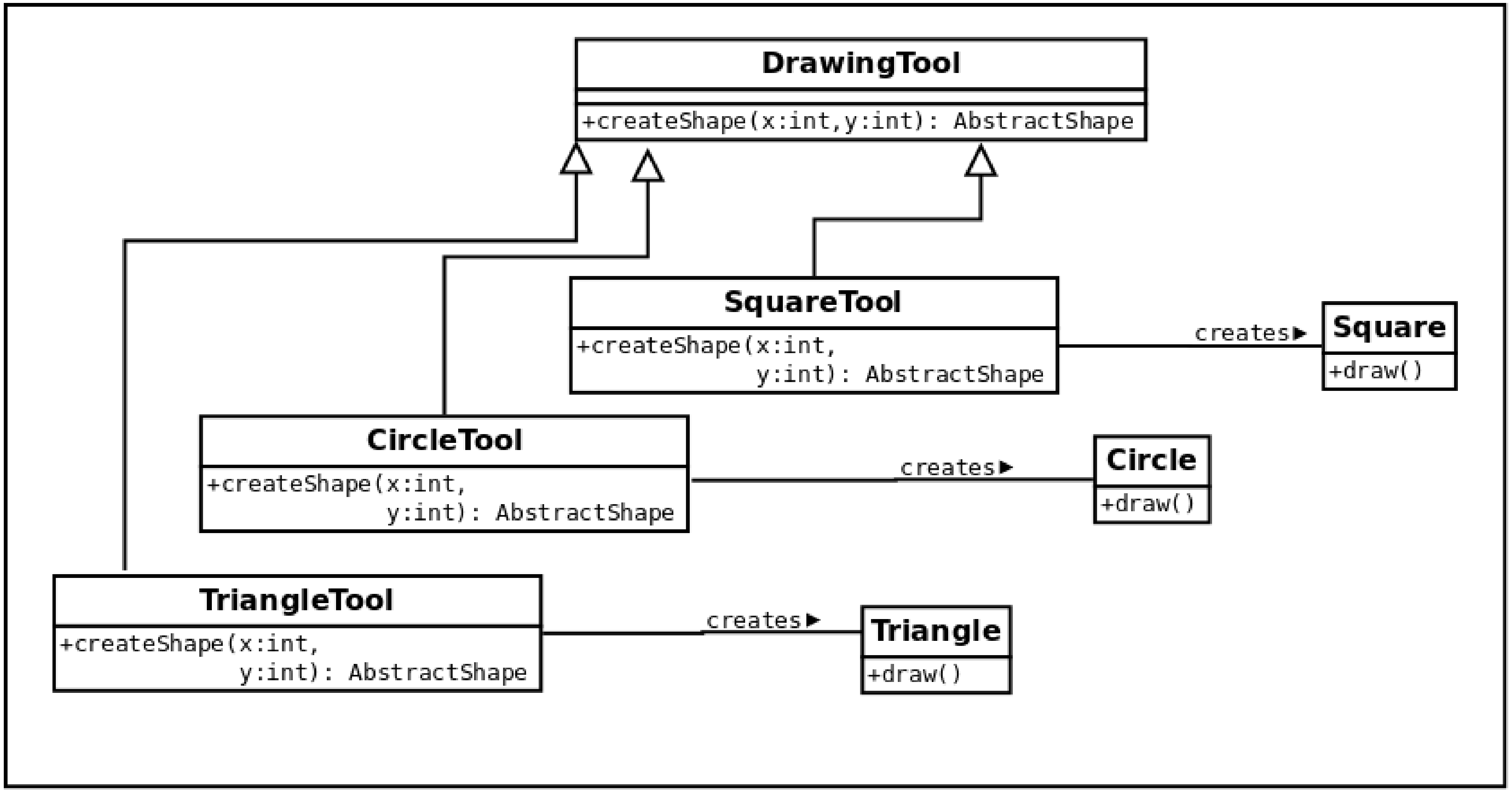


Problems:

➔***DrawingTool*** has to know about the arguments etc. for creating all types of shapes

➔Every time we add a new shape, we need to modify the ***DrawingTool*** class

Factory Method Class Diagram

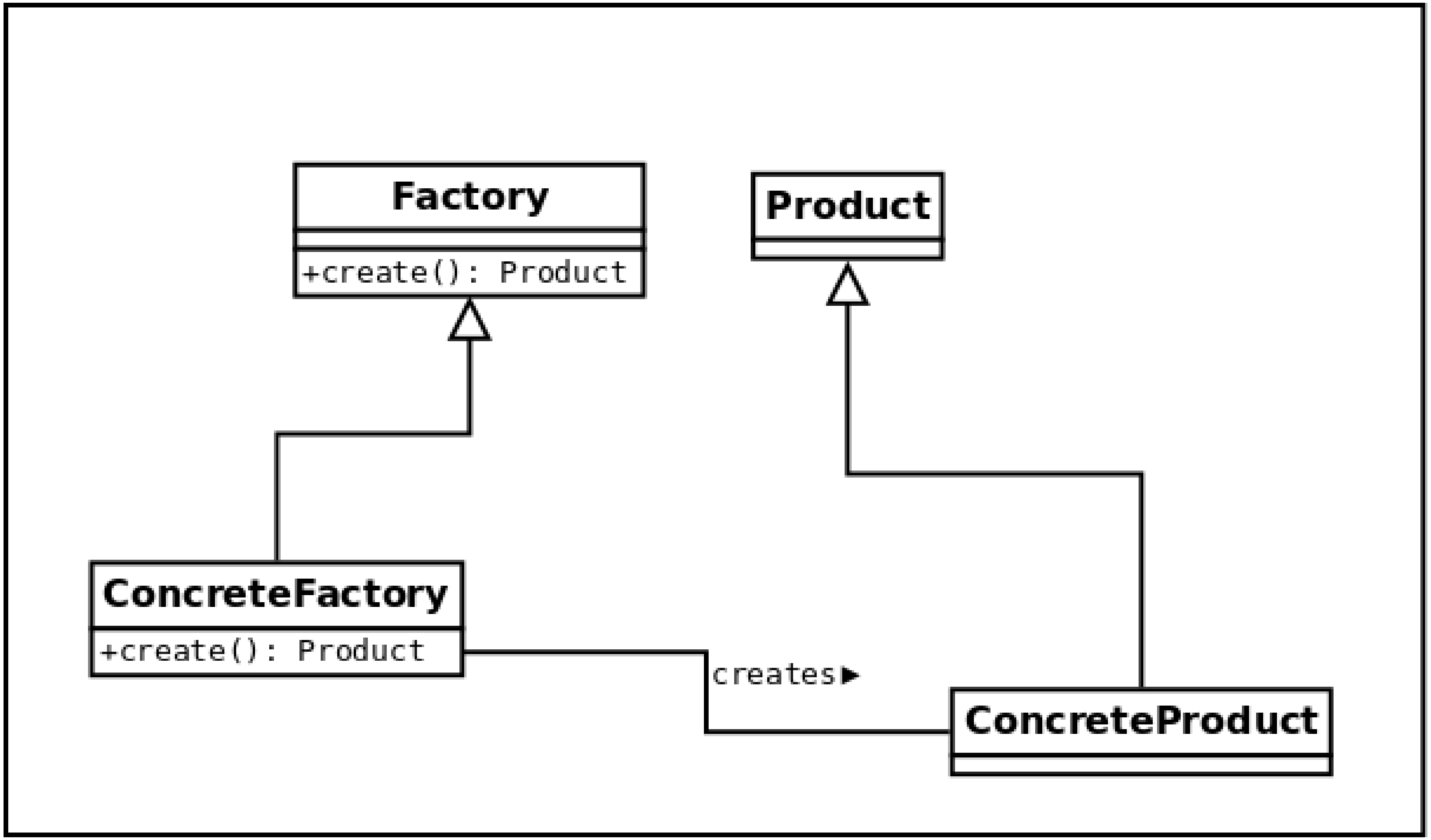

DrawingTool
+createShape(x:int,y:int): AbstractShape
SquareTool
+createShape(x:int, y:int): AbstractShape
creates▸
Square
+draw()
CircleTool
+createShape(x:int, y:int): AbstractShape
creates▸
Circle
+draw()
TriangleTool
+createShape(x:int, y:int): AbstractShape
creates▸
Triangle
+draw()

➔Clicking a tool bar icon creates or activates an instance of the appropriate subclass
➔The *createShape()* method will create the right kind of shape

➔To add a new shape, we just create a new subclass of ***DrawingTool***

- We’ve almost doubled the number of classes!
- But to evolve the system, we don’t need to change existing classes – less likely to cause bugs
- In general, using patterns may increase the number of classes – but should decrease coupling

General Pattern



Other Uses for Factory Classes

- Share instances by providing the same reference to multiple clients (e.g. *javax.swing.BorderFactory*)
- Generate appropriate instances based on an external representation such as a file (e.g. *java.security.cert.CertificateFactory*)
- Manage a pool of objects to conserve resources and save time (e.g. *DatabaseConnectionFactory*)

There are really two different (but related) patterns: FactoryMethod and Factory.

Both give the designer more control over the creation of class instances than just using constructors.

Observer Pattern

Problem

When events occur in one class, we need a way to dynamically notify instances of other classes. We do not know at design time what other classes should be notified, or when.

Solution

Create a way to register outside classes as observers of the class where events occur. All registered classes have a standard interface for notification. When events occur, all observing classes will be notified.

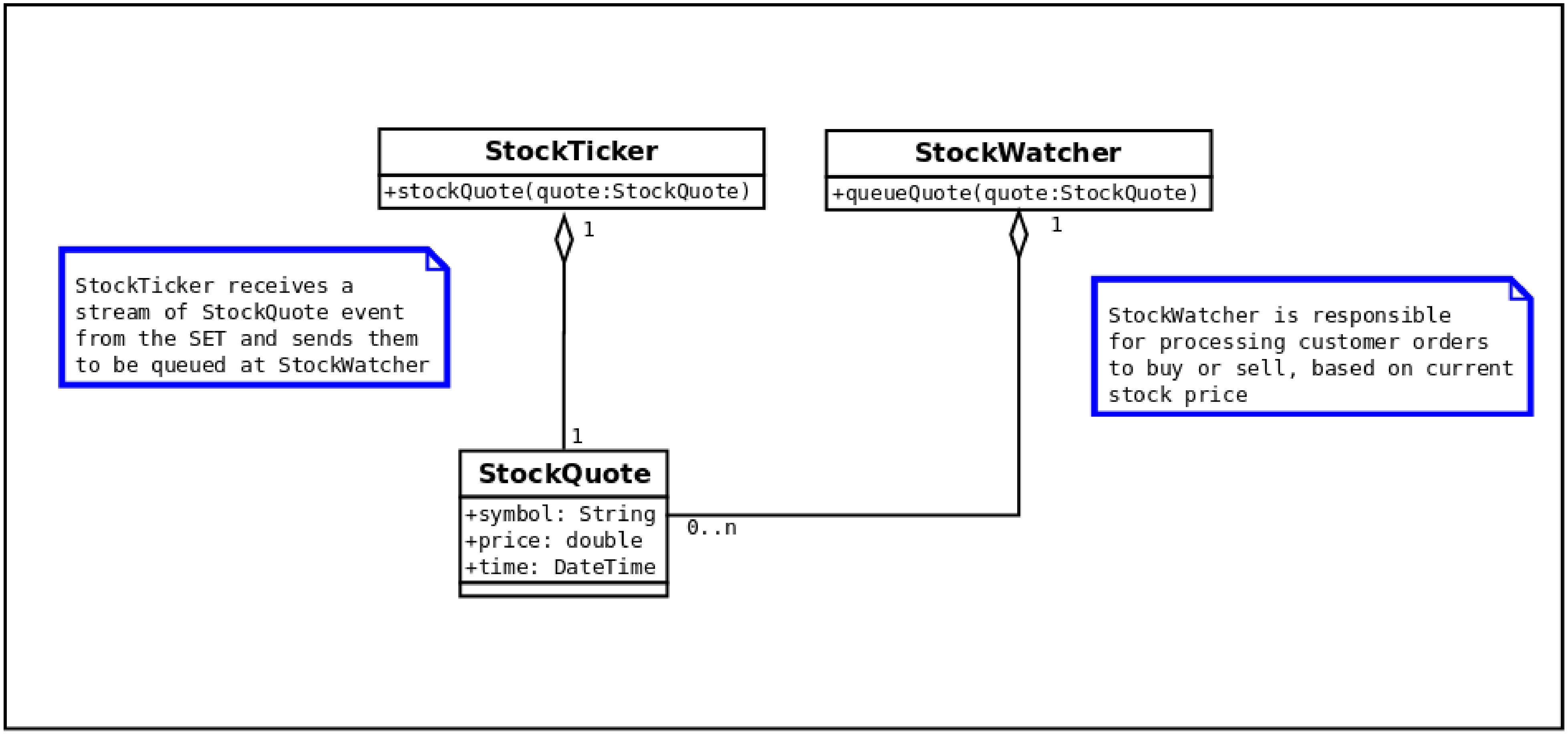
Example

Suppose we're creating a stock trading application...

Stock Trading Requirements

- Users can create orders to buy or sell specific stocks
- Orders should be triggered when the stock price reaches some specific price level.
- Stock orders can be created or deleted at any time.
- Buy/sell actions should be executed in near-real-time

Initial Class Diagram


StockTicker
+stockQuote(quote:StockQuote)
StockWatcher
+queueQuote(quote:StockQuote)
1
1
StockTicker receives a stream of StockQuote event from the SET and sends them to be queued at StockWatcher
StockWatcher is responsible for processing customer orders to buy or sell, based on current stock price
1
StockQuote
+symbol: String
+price: double
+time: DateTime
0..n

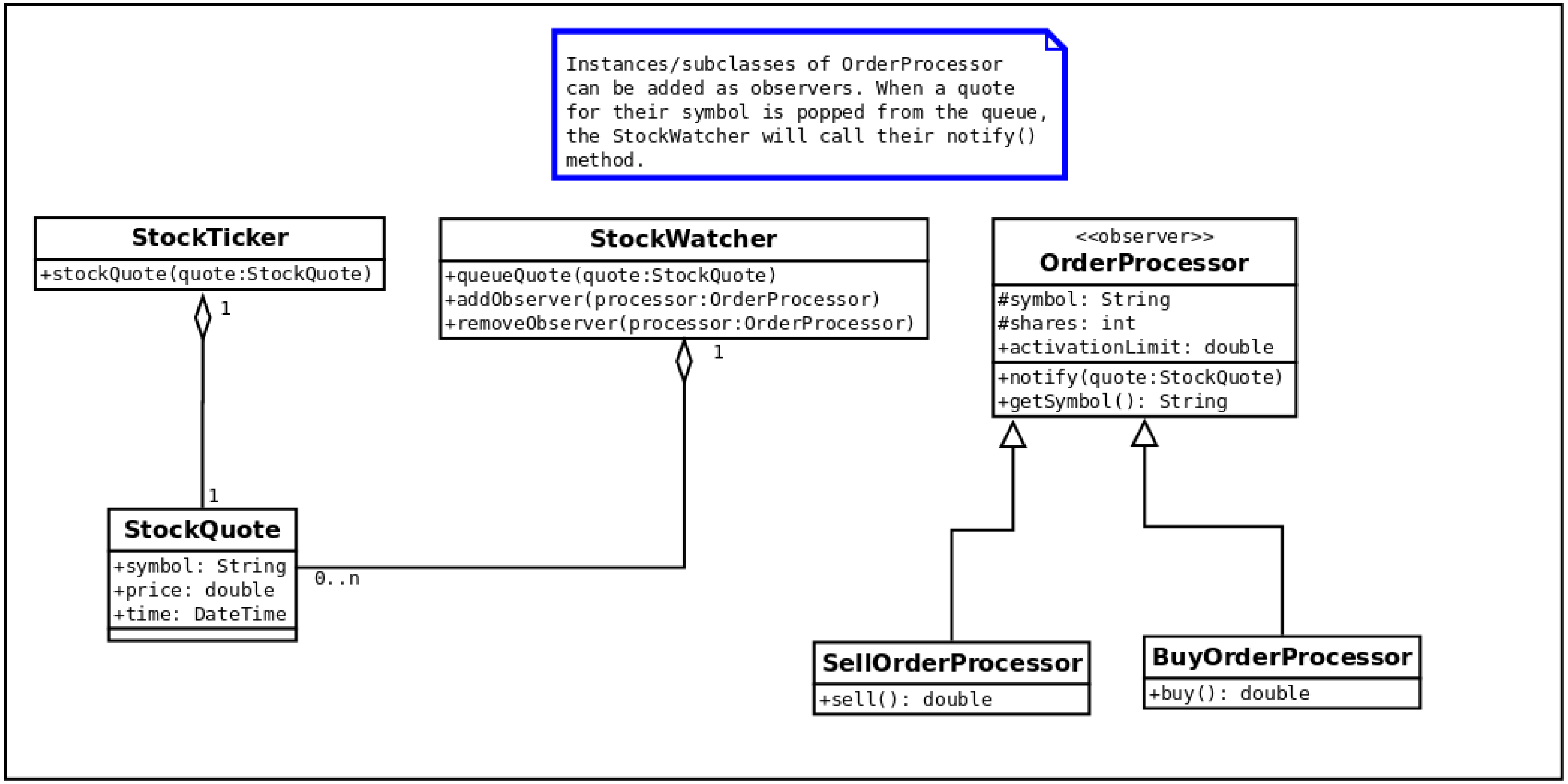
Problems:

➔***StockWatcher*** must has transfer information to "interested" parties – but who is interested?

➔***StockWatcher*** needs to act quickly to avoid being overwhelmed by data

➔New orders can be created at any time.

Using the Observer Pattern

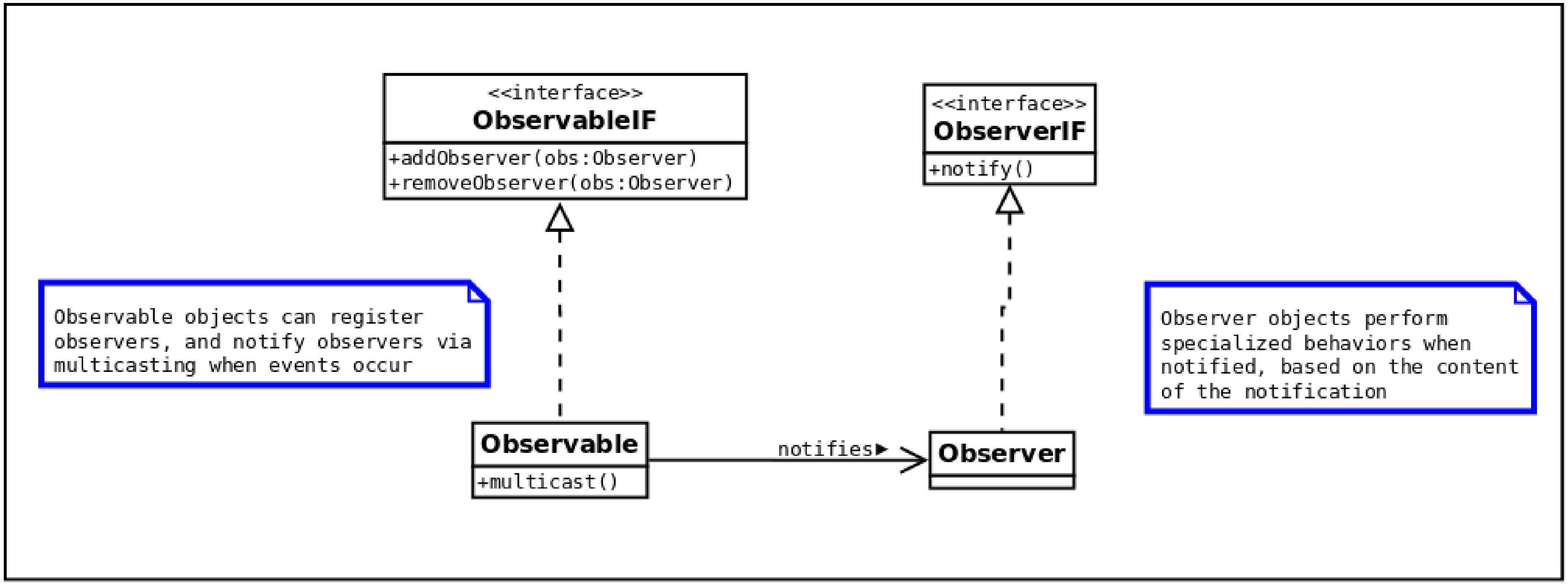

Instances/subclasses of OrderProcessor can be added as observers. When a quote for their symbol is popped from the queue, the StockWatcher will call their notify() method.
StockTicker
+stockQuote(quote:StockQuote)
1
1
StockWatcher
+queueQuote(quote:StockQuote)
+addObserver(processor:OrderProcessor)
+removeObserver(processor:OrderProcessor)
1
<<observer>>
OrderProcessor
#symbol: String
#shares: int
+activationLimit: double
+notify(quote:StockQuote)
+getSymbol(): String
StockQuote
+symbol: String
+price: double
+time: DateTime
0..n
SellOrderProcessor
+sell(): double
BuyOrderProcessor
+buy(): double

- ➔System creates an *OrderProcessor* instance for each stock order
- ➔Each *OrderProcessor* is added as an observer to *StockWatcher*
- ➔*StockWatcher* can organize observers by stock symbol
- ➔When it dequeues a *StockQuote* for a particular stock, *StockWatcher* notifies all the observers for this stock

Advantages of Observer Pattern

- Offload work from **StockWatcher** – this class now works mostly as a "dispatcher", notifying other class instances
- Can create new types of ***OrderProcessors*** that execute different or more complicated trades using different rules, without changing ***StockTicker*** or ***StockWatcher***
- Could create "null" order processors that do not buy or sell anything, but which accumulate or provide information to their creators

General Form of Observer Pattern



<<interface>>
ObservableIF
+addObserver(obs:Observer)
+removeObserver(obs:Observer)
<<interface>>
ObserverIF
+notify()
Observable objects can register observers, and notify observers via multicasting when events occur
Observer objects perform specialized behaviors when notified, based on the content of the notification
Observable
+multicast()
notifies▸
Observer

This should seem familiar! You have already worked with an important example of this pattern...

Can you remember what it is?

Adapter Pattern

Problem

Our system needs to to exchange information with or initiate behavior in an external system. The interface to the external system is already defined; we cannot change it. We do not want to make extensive changes to the design or implementation of our system.

Solution

Create a bridging class to adapt the interface of our system to that of the external system.

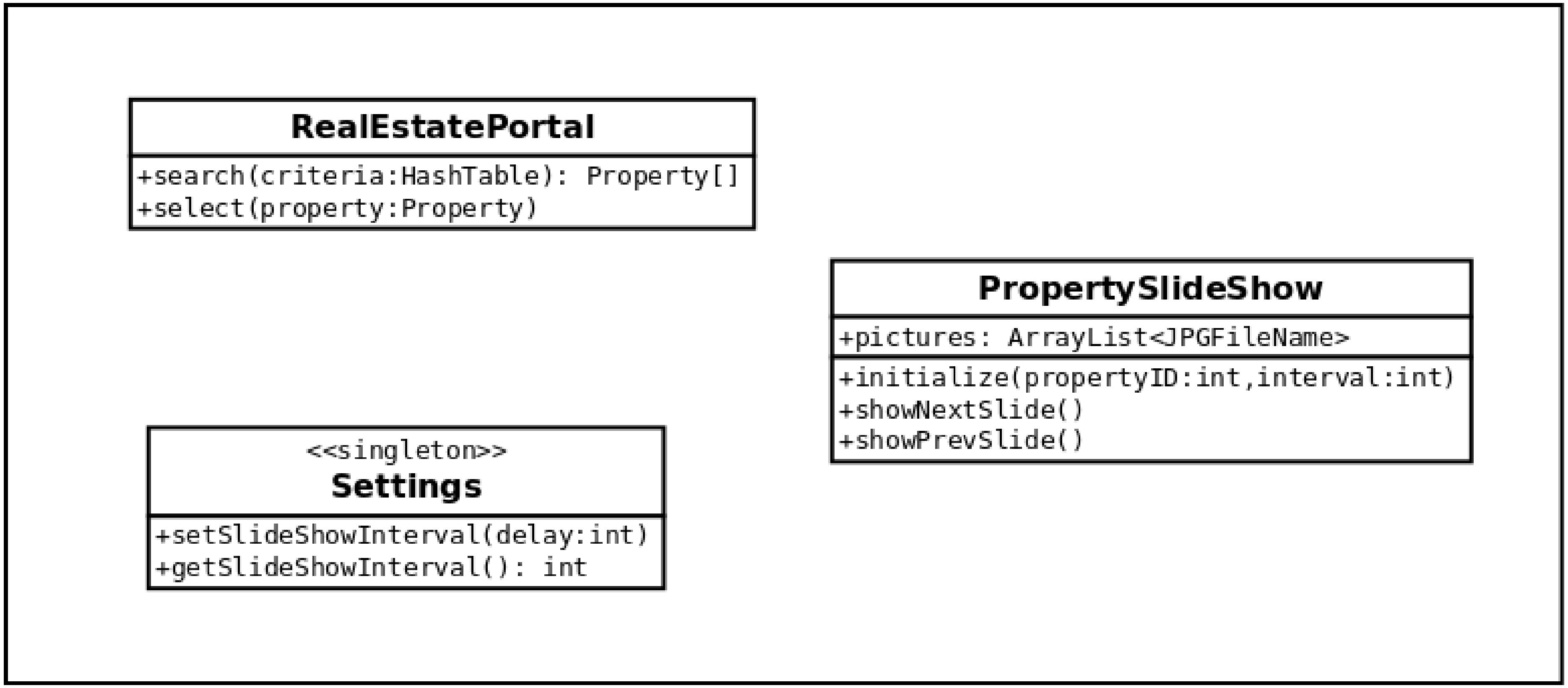
Example

Suppose we’re creating a real estate portal application...

Real Estate Portal Requirements

- Users can search for houses or condos based on a set of personal criteria
- The system displays a list of properties that match the user’s criteria (note: here “property” means house or condo, not “attribute”)
- The user selects a property to view
- The system displays information about the selected property, including an automatic slide show of photos.

Real Estate Initial Class Diagram

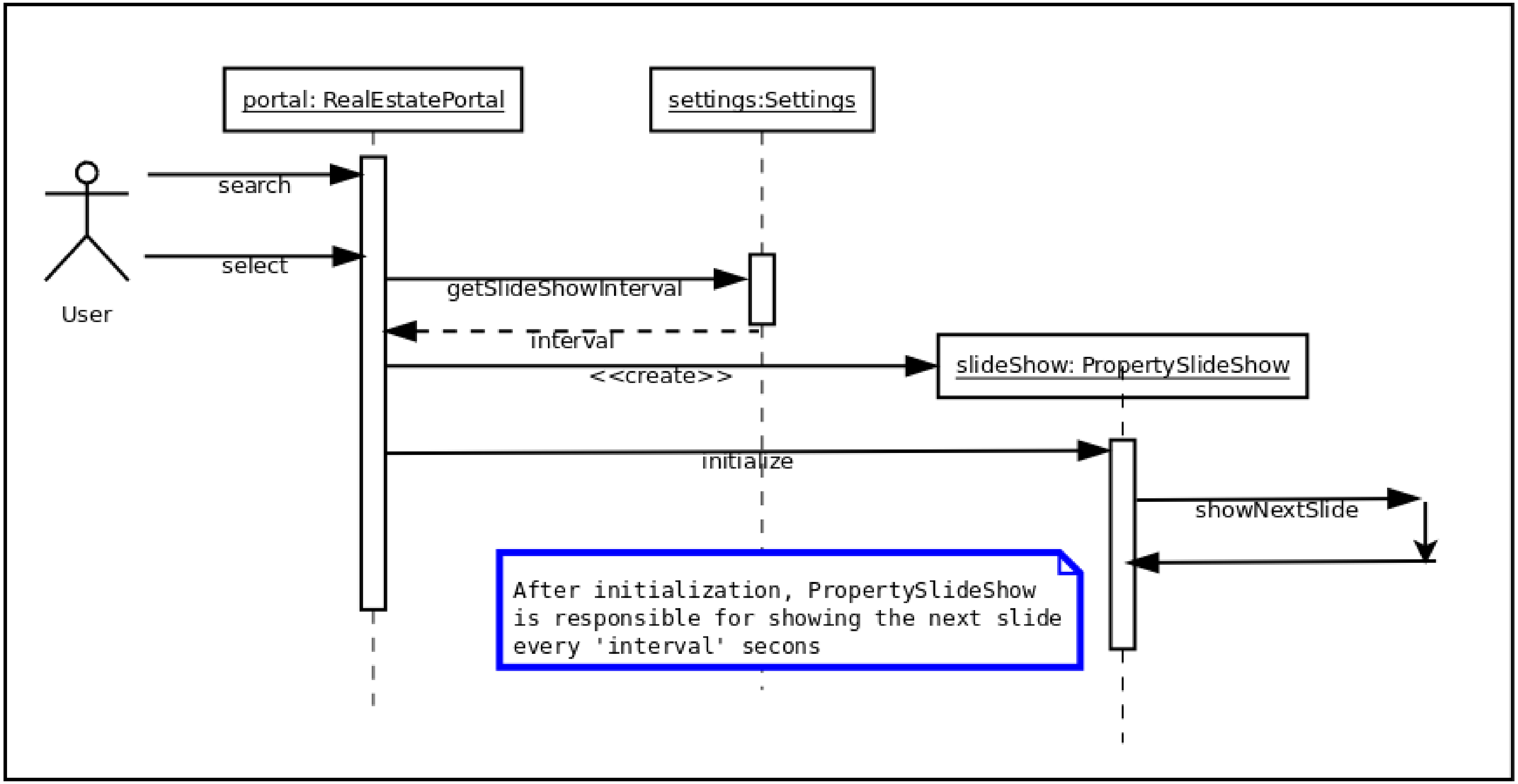

RealEstatePortal
+search(criteria:HashTable): Property[]
+select(property:Property)
PropertySlideShow
+pictures: ArrayList<JPGFileName>
+initialize(propertyID:int,interval:int)
+showNextSlide()
+showPrevSlide()
<<singleton>>
Settings
+setSlideShowInterval(delay:int)
+getSlideShowInterval(): int

➔*PropertySlideShow* encapsulates both access to and display of the photos

➔*Settings* lets the user control the speed of the slide show

➔*RealEstatePortal* provides the top level UI for searching and selection

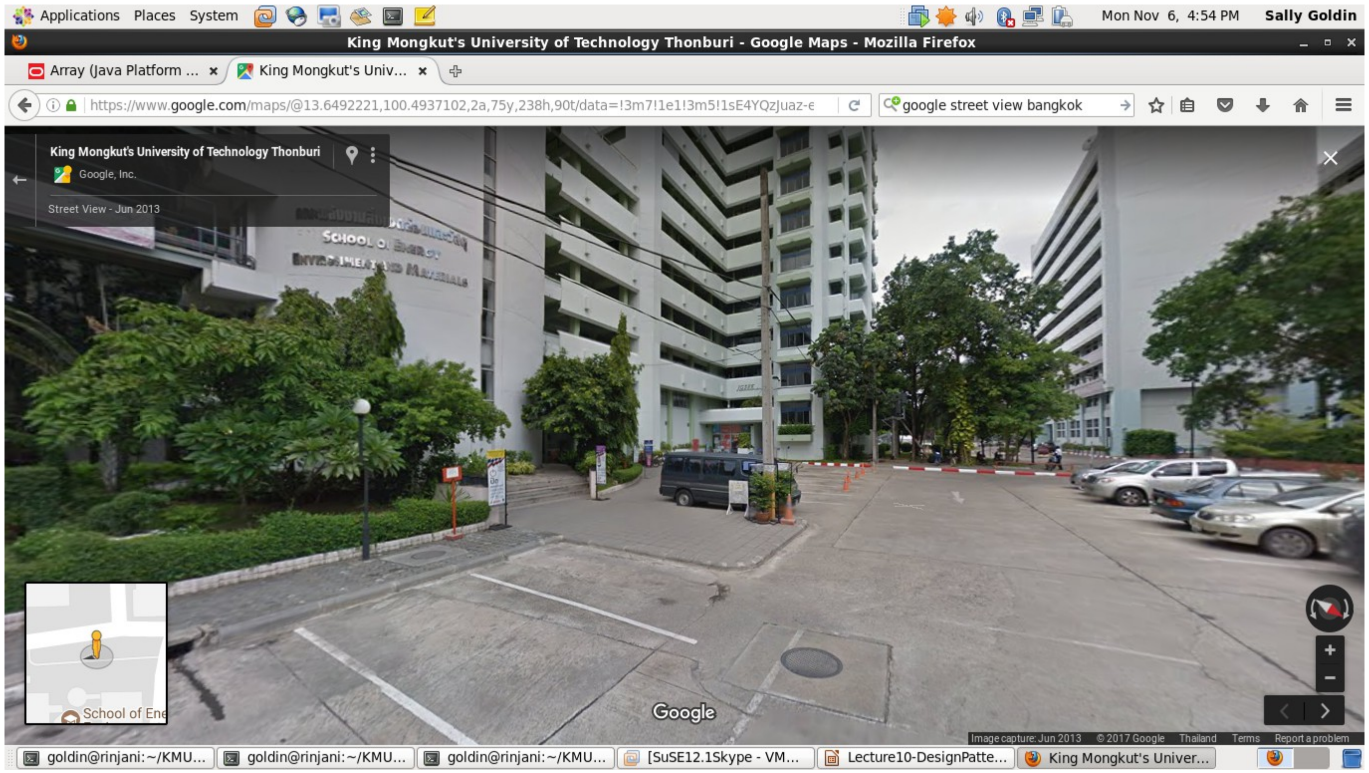
Real Estate Sequence Diagram



After initialization, PropertySlideShow is responsible for showing the next slide every 'interval' secons

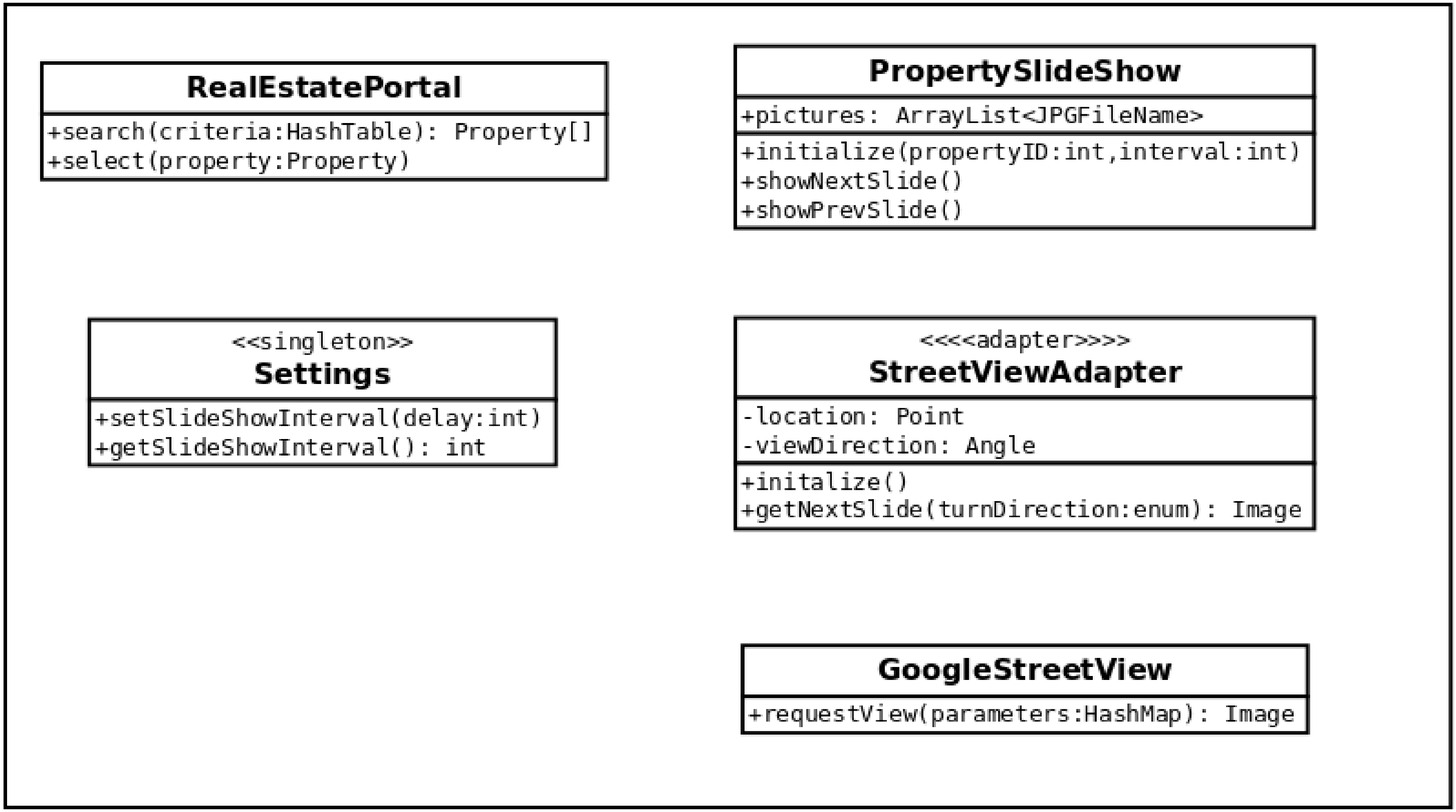
New Requirement!

Display a slide show of the surrounding area using Google StreetView

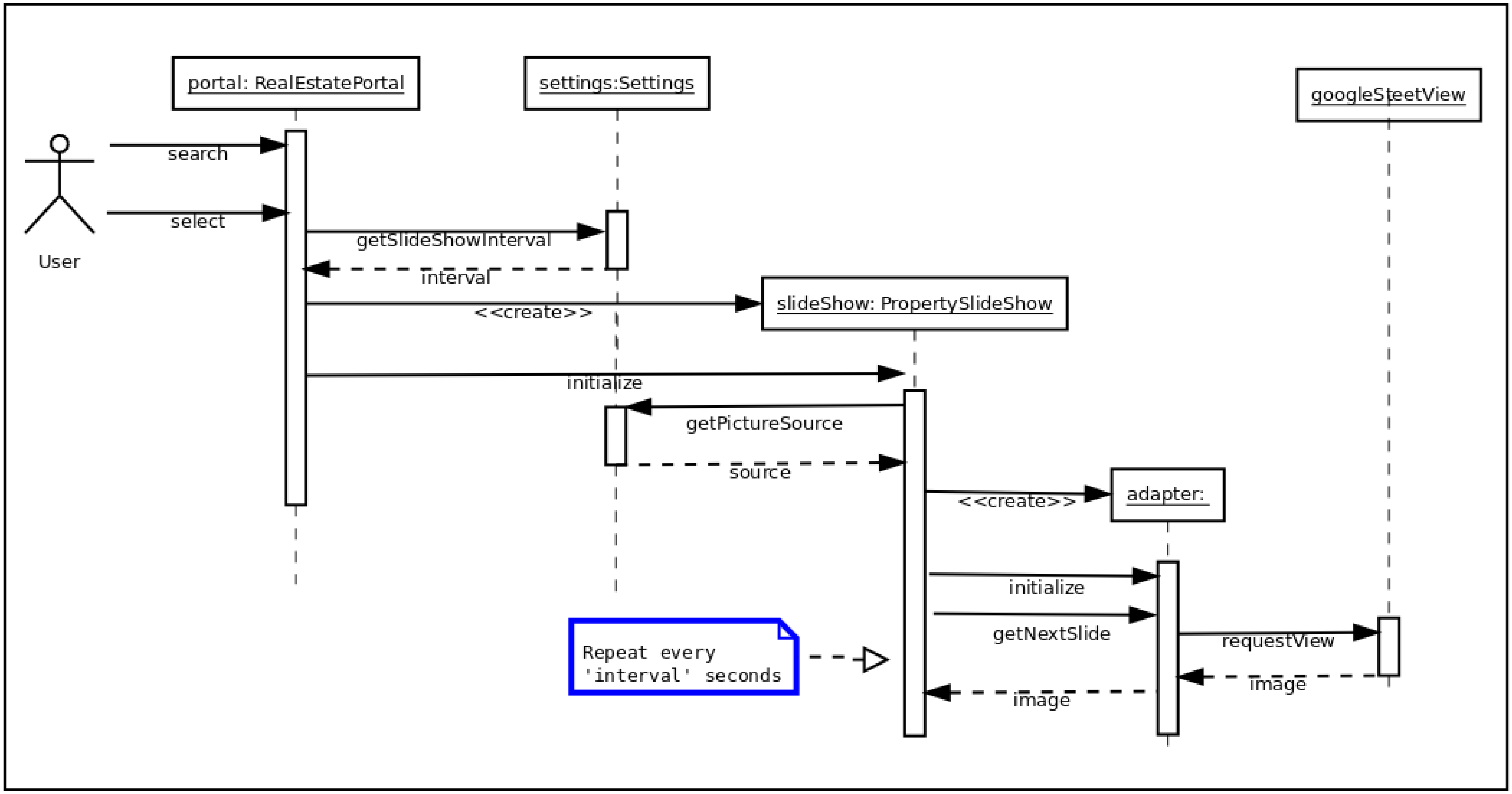

Applications Places System
Mon Nov 6, 4:54 PM Sally Goldin
King Mongkut's University of Technology Thonburi - Google Maps - Mozilla Firefox
Array (Java Platform ...
King Mongkut's Univ...
https://www.google.com/maps/@13.6492221,100.4937102,2a,75y,238h,90t/data=!3m7!1e1!3m5!1sE4YQzJuaz-e
google street view bangkok
King Mongkut's University of Technology Thonburi
Google, Inc.
Street View - Jun 2013
School of Ene
Google
Image capture: Jun 2013 © 2017 Google Thailand Terms Report a problem
goldin@rinjani:~/KMU...
goldin@rinjani:~/KMU...
goldin@rinjani:~/KMU...
[SuSE12.1Skype - VM...
Lecture10-DesignPatte...
King Mongkut's Univer...

To get images from StreetView, we need to access a web service and use Google's API

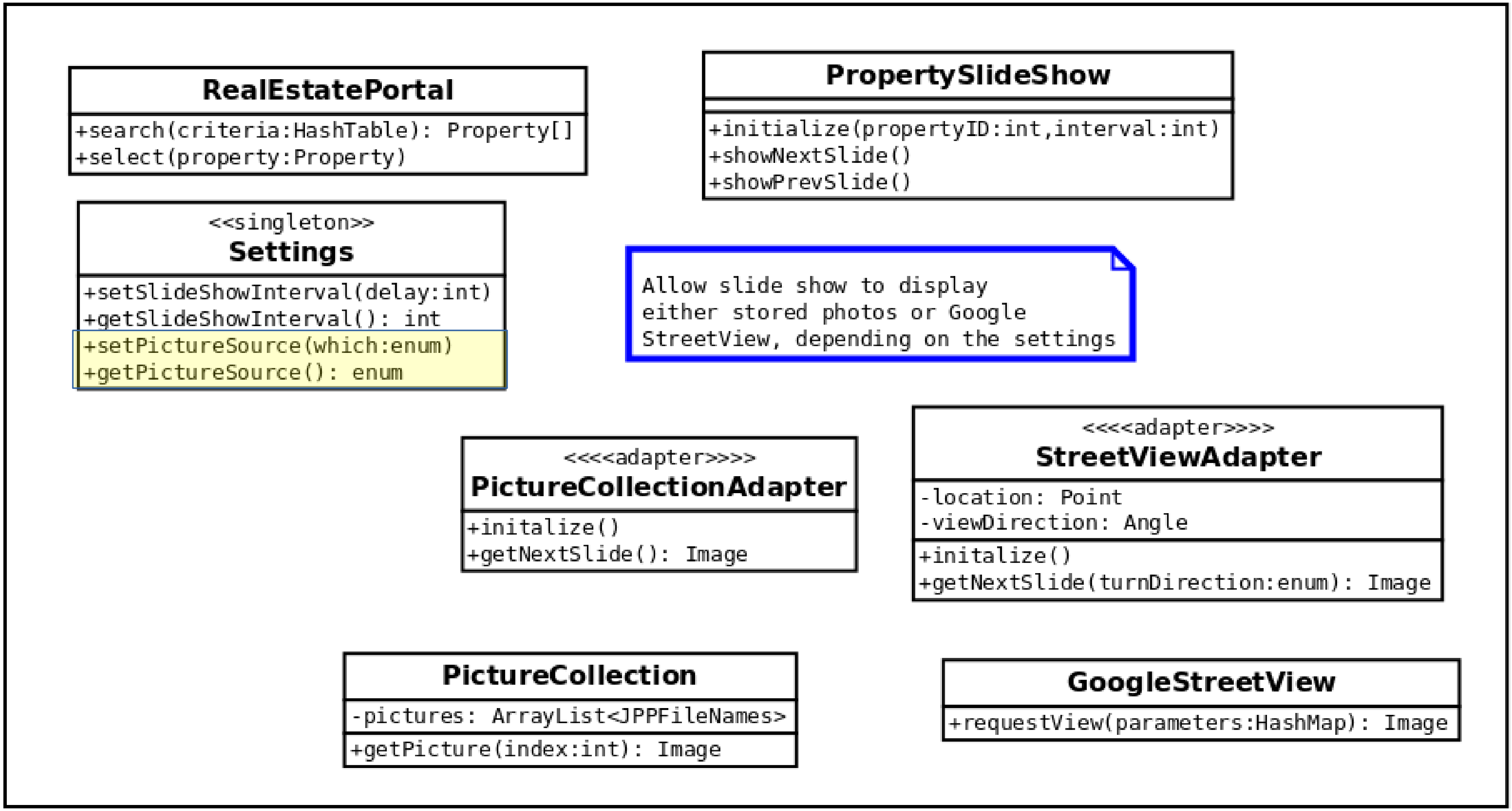
Class Diagram using Adapter


RealEstatePortal
+search(criteria:HashTable): Property[]
+select(property:Property)
PropertySlideShow
+pictures: ArrayList<JPGFileName>
+initialize(propertyID:int,interval:int)
+showNextSlide()
+showPrevSlide()
<<singleton>>
Settings
+setSlideShowInterval(delay:int)
+getSlideShowInterval(): int
<<<<adapter>>>>
StreetViewAdapter
-location: Point
-viewDirection: Angle
+initalize()
+getNextSlide(turnDirection:enum): Image
GoogleStreetView
+requestView(parameters:HashMap): Image

Sequence Diagram using Adapter



Letting the User Choose


RealEstatePortal
+search(criteria:HashTable): Property[]
+select(property:Property)
PropertySlideShow
+initialize(propertyID:int,interval:int)
+showNextSlide()
+showPrevSlide()
<<singleton>>
Settings
+setSlideShowInterval(delay:int)
+getSlideShowInterval(): int
+setPictureSource(which:enum)
+getPictureSource(): enum
Allow slide show to display either stored photos or Google StreetView, depending on the settings
<<<<adapter>>>>
PictureCollectionAdapter
+initalize()
+getNextSlide(): Image
<<<<adapter>>>>
StreetViewAdapter
-location: Point
-viewDirection: Angle
+initalize()
+getNextSlide(turnDirection:enum): Image
PictureCollection
-pictures: ArrayList<JPPFileNames>
+getPicture(index:int): Image
GoogleStreetView
+requestView(parameters:HashMap): Image

Applying Adapter More Broadly

- Allows the user to choose either stored photos or Google
- Makes it possible to provide additional image sources
- Note that we have extracted some functionality from *PropertySlideShow* and moved it to a new class, *PictureCollection*
- This is an example of **refactoring**. We'll talk more about refactoring next week.

Model-View-Controller Pattern

Problem

We need to provide a flexible, configurable user interface. Alternatively, we may need to provide different user interfaces in different situations or on different platforms. Hence we want to reduce dependencies between the user interface and the rest of the system.

Solution

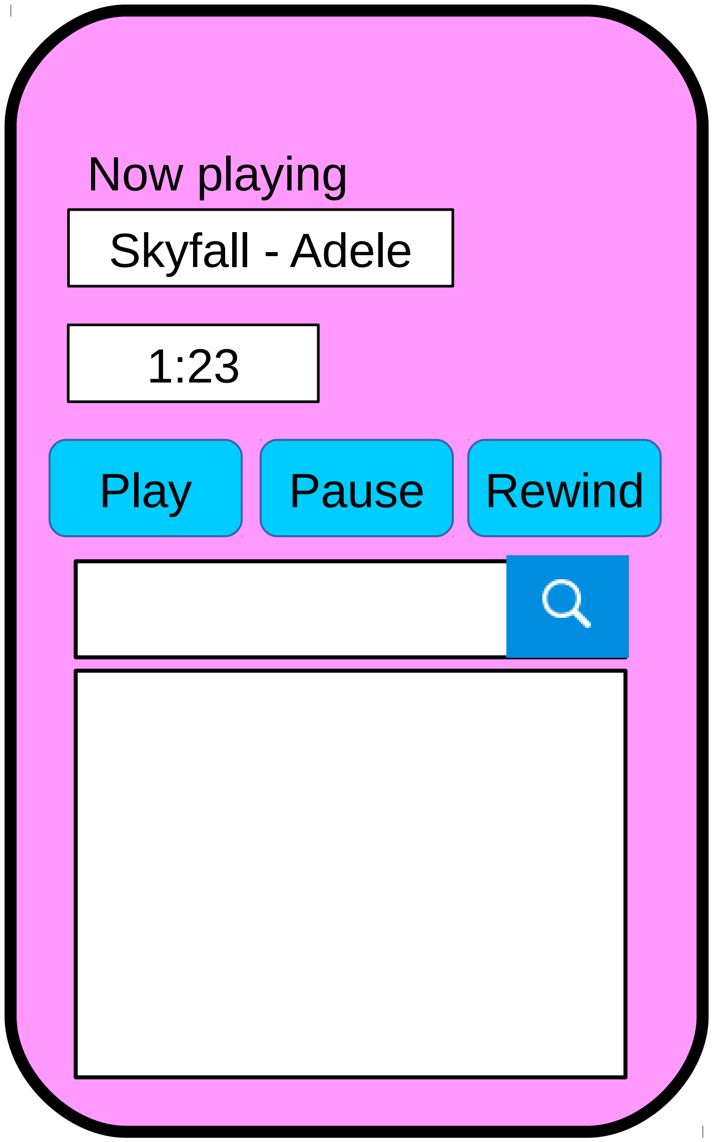
Separate the system into three main components. The ***View*** includes all the UI elements. The ***Model*** maintains knowledge and state information. The ***Controller*** reacts to changes in the Model by updating the View and executes requests from the View by changing the Model.

Example

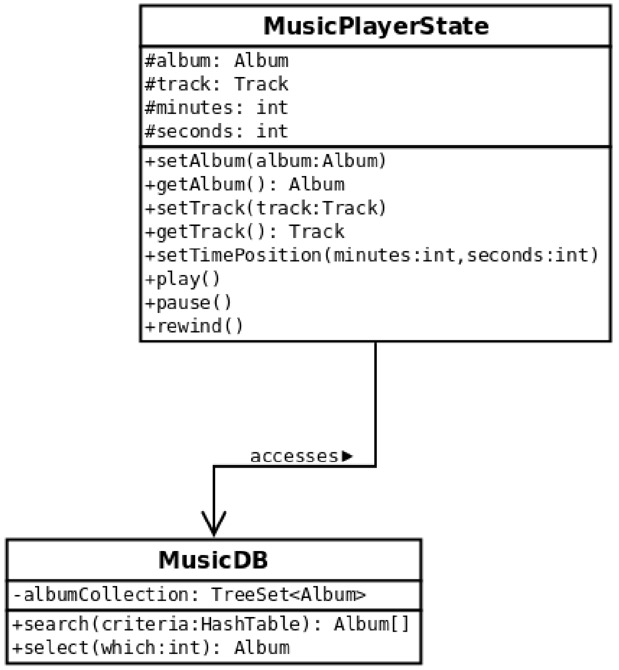
Suppose we’re creating a music player application...

Model-View-Controller (MVC) is not a GOF pattern
It is an “architectural pattern” with a larger scope

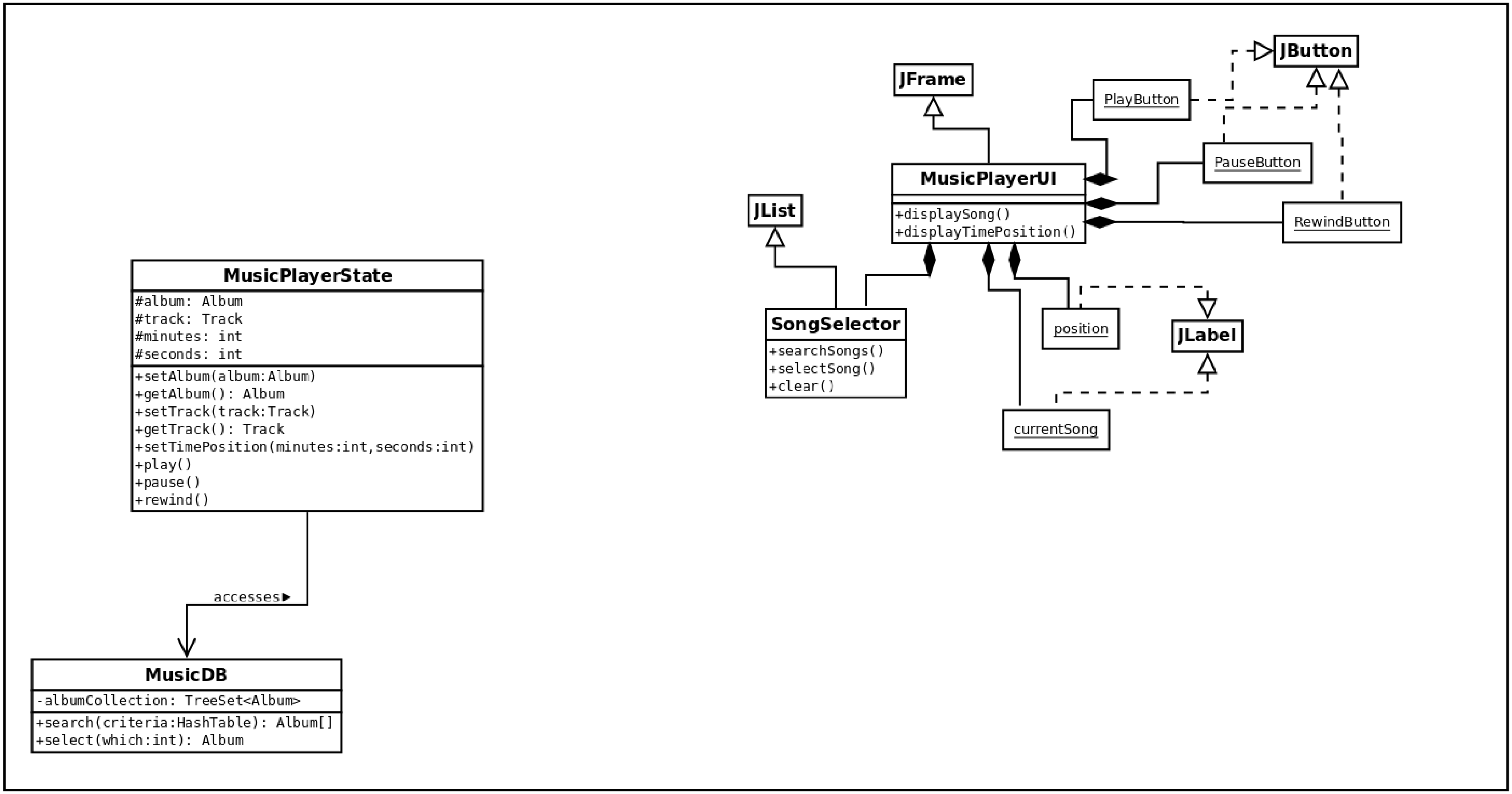
Music Player Prototype UI



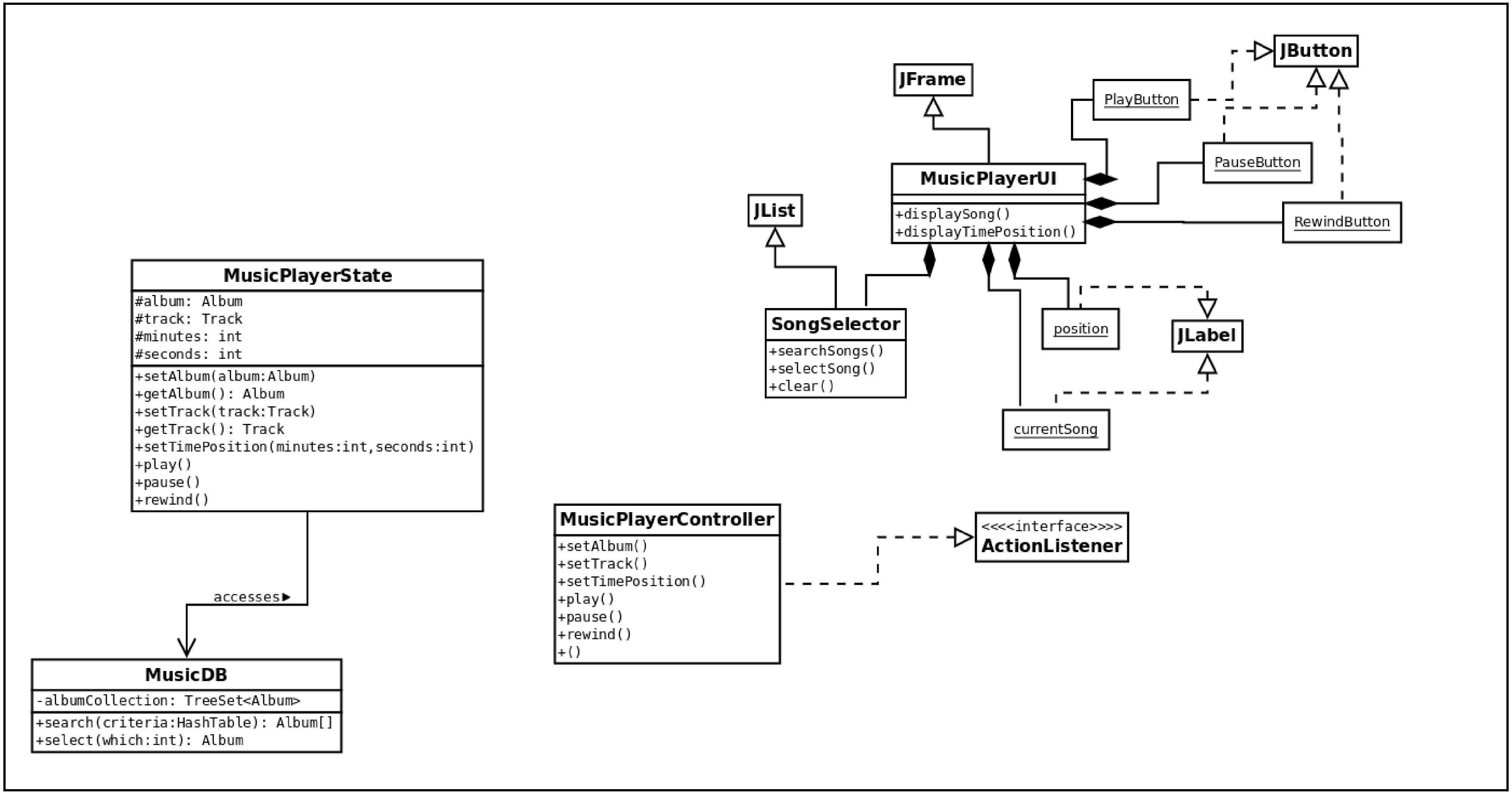
Model Classes for Music Player



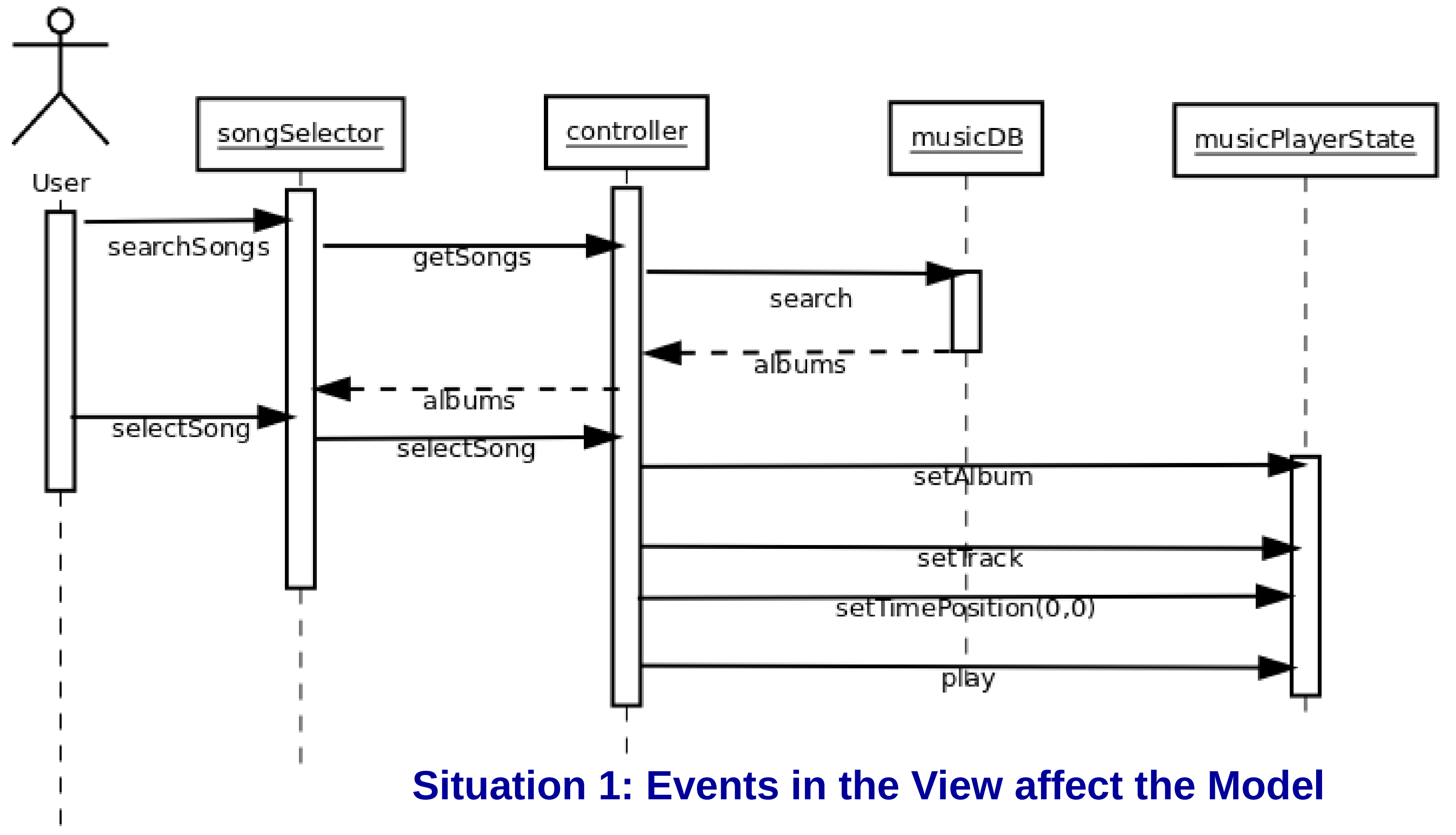
View Classes for Music Player



Controller connects Model and View

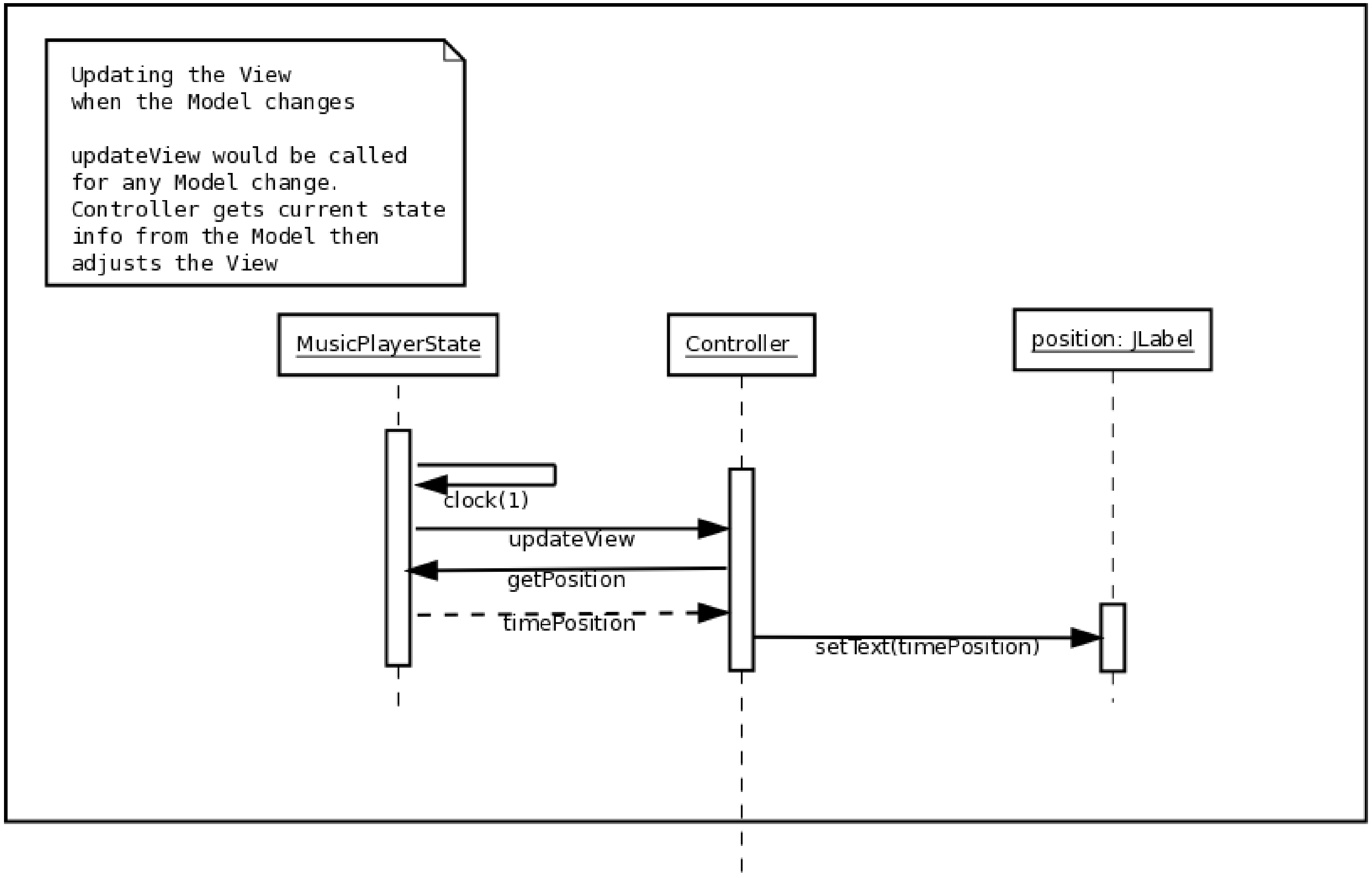


Music Player Sequence Diagram: 1



Situation 1: Events in the View affect the Model

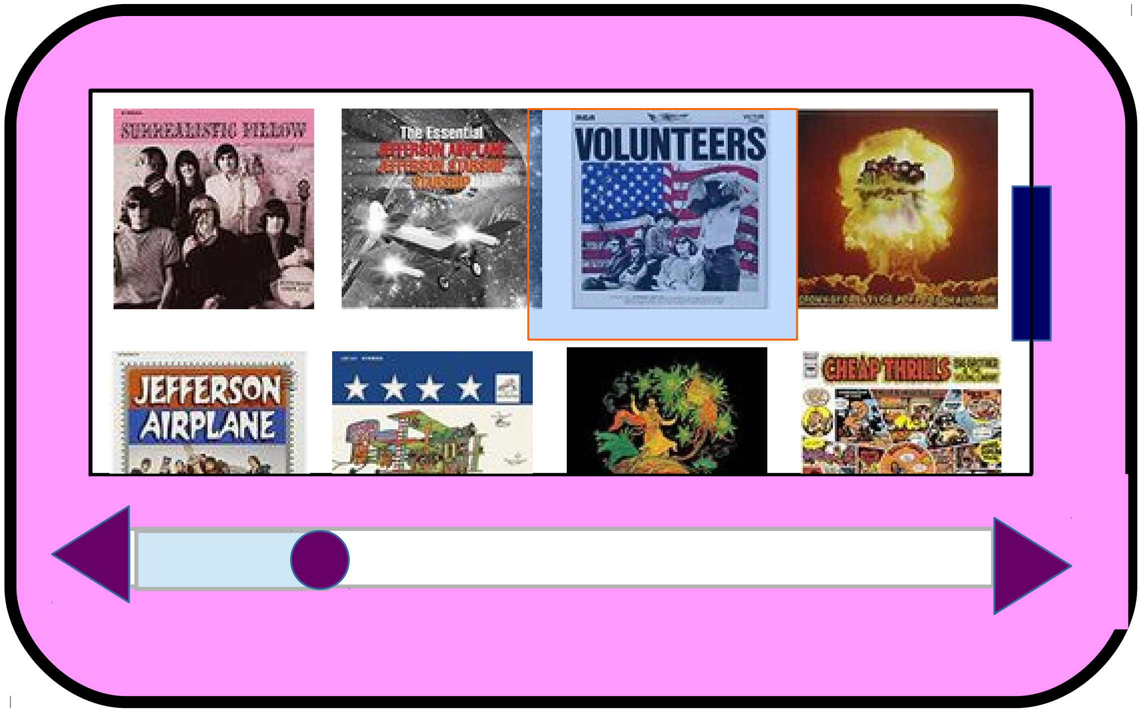
Music Player Sequence Diagram: 2



Situation 2: Events in the Model affect the View

Wouldn't it be simpler to have the UI talk directly to the model classes?

Music Player Prototype UI – Version 2


SURREALISTIC PILLOW
The Essential Jefferson Airplane Jefferson Starship Starship
VOLUNTEERS
JEFFERSON AIRPLANE
CHEAP THRILLS

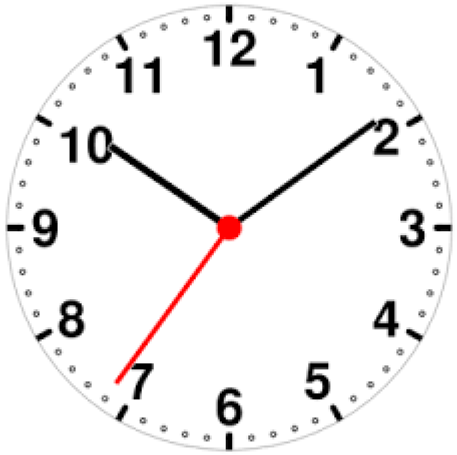
No more buttons!
No text fields or lists
Control the playback by swiping, tapping – double tap to pause

If the UI and the model were communicating directly, many changes would be needed!

Exercise

Suppose we want to create a clock app with several different possible displays:

- Analog
- Digital, 12 hours with AM/PM
- Digital, 24 hours
- Digital, 24 hours with seconds shown



9:30 PM

21:30

21:30:17

Exercise Continued

The clock app offers the following capabilities:

- Set display type (one of the four options on previous page)
- Set time (user enters current time)
- Update time based on system clock (each second)
- Set alarm time (absolute time)
- Deliver alarm

1) Create a class diagram for this app that uses the **Model-View-Controller** pattern

2) Create sequence diagrams for the **Set Time** and **Update Time** use cases that use the classes in the diagram

Exercise Continued

- Do this exercise with your project teammate
- Submit diagrams in printed form, one set for each team, next week

Please be sure to include your team name and the members' names and student IDs on all pages