MVC and Android

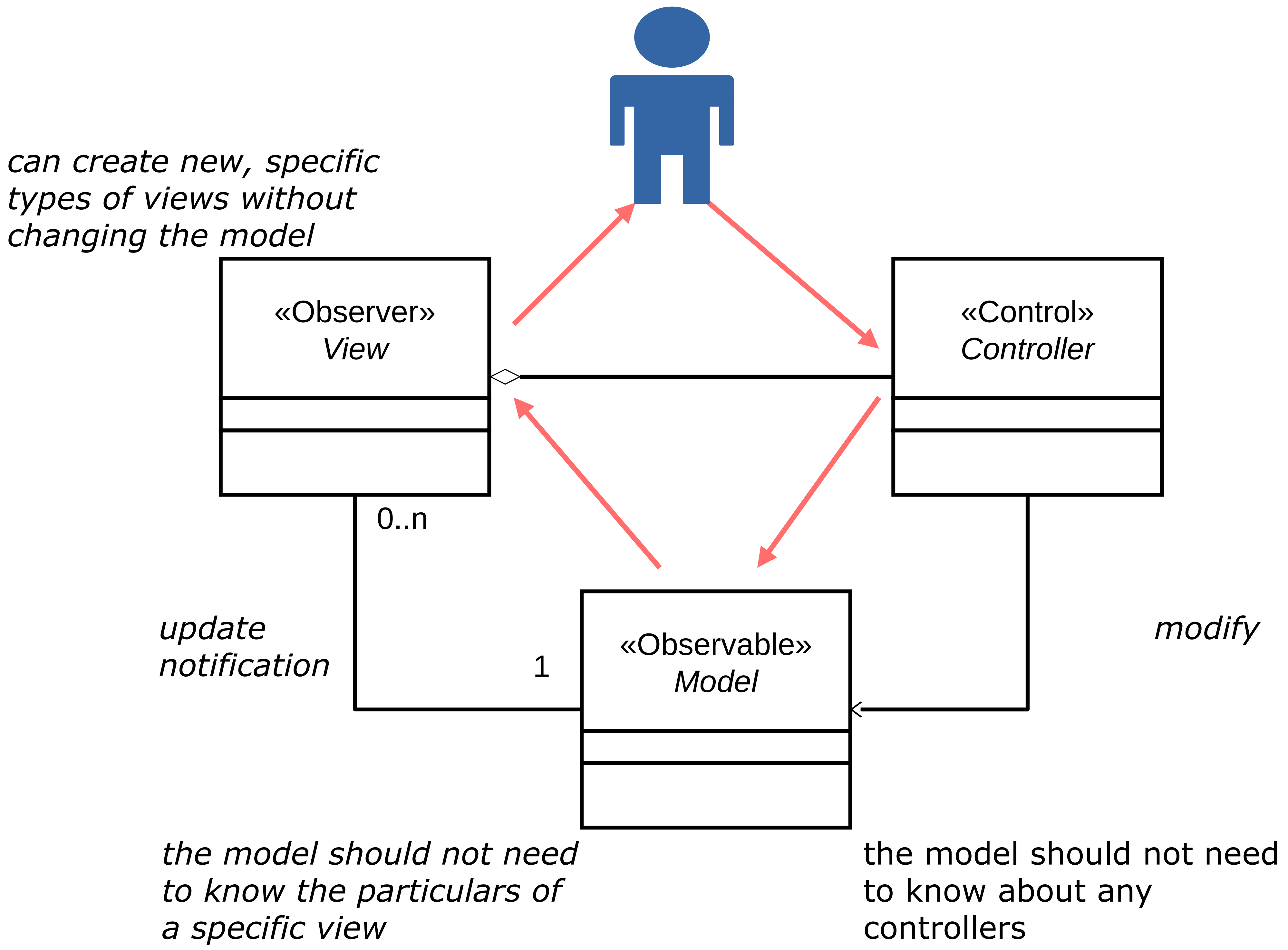
Dr. Hazel Campbell

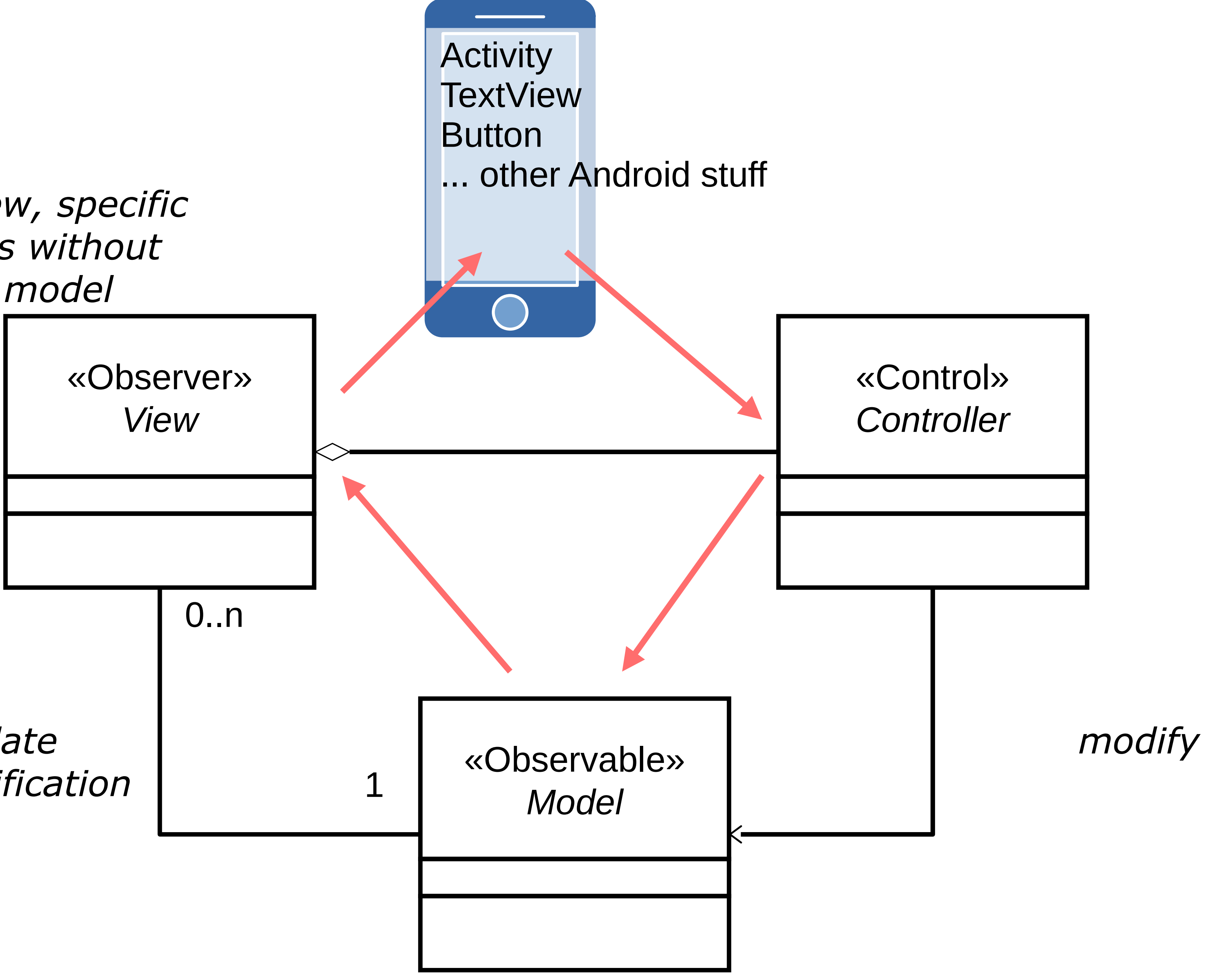
& Dr. Abram Hindle

& Dr. Ken Wong

Department of Computing Science

University of Alberta


can create new, specific types of views without changing the model
«Observer» View
«Control» Controller
0..n
update notification
1
«Observable» Model
modify
the model should not need to know the particulars of a specific view
the model should not need to know about any controllers



can create new, specific types of views without changing the model

update notification

modify

the model should not need to know the particulars of a specific view

the model should not need to know about any controllers

MVC Classes

- Model Classes
  - Entities (“Book”, “Library”, “Password”)
  - complete, self-contained representation of the data managed by the application
  - What you would save!
  - Evolves over time
  - Enforces consistency and validity rules
  - Lots of setters and getters!

MVC Classes

- Model Classes
  - Enforces consistency and validity rules
    - Enforce multiplicities
    - Enforce composition relationship
    - Enforce positive numbers are positive
    - Enforce passwords have at least one number and a special symbol…

MVC Classes

- Model Classes
  - These classes get saved to file…
    - Or stored in database
    - Or mapped to database with ORM
    - Or sent over the internet

- Has add/remove(observer) methods.
- Calls update(this) on all the observers when the model data changes!

MVC Classes

- View Classes
  - Boundary Layer
  - Presentation Layer
  - Talk to the toolkit
    - Android: Activities, Fragments, ListViews, etc.
  - Main responsibility
    - Presentation issues!

MVC Classes

- View Classes
  - Main responsibility
    - Presentation issues!
      - Format a timestamp 1727381284 -> “Thu 26 Sep 2024 02:08:04 PM MDT”
      - Make sure lists are sorted if we want them sorted in the UI
      - person.getFirstName() + “ “ + person.getLastName()

MVC Classes

- View Classes
  - Has a reference to the Model class so it can call lots of getters.
  - model.addView(this) in the constructor.
    - (Observer pattern!)

MVC Classes

- Controller Classes
  - Boundary Layer
  - Presentation Layer
  - Handle Events
  - Talk to the toolkit
    - Android: Activities, Fragments, ListViews, etc.
  - Main responsibility
    - Transforming input into the format the Model wants

MVC Classes

- Controller Classes
  - Main responsibility
    - Transforming input into the format the Model wants
    - Parse a timestamp “Thu 26 Sep 2024 02:08:04 PM MDT” -> 1727381284
    - Split firstName and lastName

MVC Classes

- Controller Classes
  - Has a reference to the Model class so it can call lots of setters.
  - Generally constructed with the model it’s meant to control.

Observer Pattern

```
public abstract class AbstractObservable {
    private final transient Set<AbstractObserver> observers;

    protected AbstractObservable() {
        observers = new ArraySet<>();
    }

    public void addObserver(AbstractObserver observer) {
        observers.add(observer);
        observer.update(this);
    }

    public void removeObserver(AbstractObserver observer) {
        observers.remove(observer);
    }

    public void notifyObservers() {
        // Call this from setters
        for (AbstractObserver observer : observers) {
            observer.update(this);
        }
    }
}
```

Observer Pattern

```
public abstract class AbstractObserver {
    public transient AbstractObservable observable;

    public void startObserving(AbstractObservable observable) {
        // call me from the constructor or when ready
        if (this.observable != null) {
            throw new RuntimeException("Can't view two models!");
        }
        this.observable = observable;
        observable.addObserver(this);
    }

    public void stopObserving() {
        // call me from delete() or close() etc.
        observable.removeObserver(this);
        this.observable = null;
    }

    public abstract void update(AbstractObservable whoUpdatedMe);
}
```

Abstract Model Class

```
public abstract class AbstractModel {
    private final transient Set<AbstractView> views;

    protected AbstractModel() {
        views = new ArraySet<>();
    }

    public void addView(AbstractView view) {
        views.add(view);
        view.update(this);
    }

    public void removeView(AbstractView view) {
        views.remove(view);
    }

    public void notifyViews() {
        // Call this from setters
        for (AbstractView view : views) {
            view.update(this);
        }
    }
}
```

Abstract View Class

```
public abstract class AbstractView {
    private AbstractModel model;

    public void startObserving(AbstractModel model) {
        // called during the constructor ...
        // ... or when its ready to start getting updates
        if (this.model != null) {
            throw new RuntimeException("Can't view two models!");
        }
        this.model = model;
        model.addView(this);
    }

    public void closeView() {
        // when the view goes away
        model.removeView(this);
        this.model = null;
    }

    public abstract void update(AbstractModel whoUpdatedMe);

    public AbstractModel getModel() {
        return model;
    }
}
```

Abstract Controller Class

```
public abstract class AbstractController {
    private final AbstractModel model;

    public AbstractController(AbstractModel model) {
        this.model = model;
    }

    public AbstractModel getModel() {
        return model;
    }
}
```

Concrete Model Class

```
public class TimerModel extends AbstractModel {
    /* this is our actual model stuff */
    private long startTime;
    private long duration;
    private boolean running;

    public void updateDuration() {
        duration = System.currentTimeMillis() - startTime;
        notifyViews();
    }
    public long getDuration() {
        return duration;
    }
    public boolean isRunning() {
        return running;
    }
    public void startTimer() {
        if (running) {
            throw new RuntimeException("Already running!");
        }
        startTime = System.currentTimeMillis();
        running = true;
        updateDuration();
        scheduleUpdate();
    }
    public void stopTimer() {
        if (!running) {
            throw new RuntimeException("Not running!");
        }
        running = false;
        scheduled.cancel(false);
        notifyViews();
    }
```

Concrete Model Class - Evolution

```
  /* This stuff is just for updating the duration over time (evolution)
*/
    private ScheduledFuture<?> scheduled;
    private final ScheduledExecutorService scheduler  =
Executors.newScheduledThreadPool(1);;
    private final Runnable updater = new Runnable() {
        @Override
        public void run() {
            updateDuration();
            if (running) {
                scheduleUpdate();
            }
        }
    };

    private void scheduleUpdate()
    {
        scheduled = scheduler.schedule(updater, 1, TimeUnit.SECONDS);
    }
}
```

Concrete View Class

```java
public class TimerView extends AbstractView {
    private final MainActivity mainActivity;

    public TimerView(TimerModel model, MainActivity mainActivity) {
        this.mainActivity = mainActivity;
        this.startObserving(model);
    }
    @Override
    public TimerModel getModel() {
        // Dangerous downcast!
        return (TimerModel) super.getModel();
    }
    @Override
    public void update(AbstractModel whoUpdatedMe) {
        if (getModel().isRunning()) {
            long duration = getModel().getDuration();
            long ms = duration % 1000;
            long seconds = duration / 1000 % 60;
            long minutes = duration / 1000 / 60;
            String stringDuration =
                   String.format("%dm %02ds %03dms", minutes, seconds, ms);
            mainActivity.showDuration(stringDuration);
        } else {
            mainActivity.showDuration("Timer Stopped");
        }
    }
}
```

Concrete Controller Class

```
public class TimerController extends AbstractController {
    public TimerController(TimerModel model) {
        super(model);
    }

    @Override
    public TimerModel getModel() {
        return (TimerModel) super.getModel();
    }

    public void stopButtonPressed() {
        if (getModel().isRunning()) {
            getModel().stopTimer();
        }
    }

    public void startButtonPressed() {
        if (!getModel().isRunning()) {
            getModel().startTimer();
        }
    }
}
```

Android Activity

```
public class MainActivity extends AppCompatActivity {

    private AppBarConfiguration appBarConfiguration;
    private ActivityMainBinding binding;
    private TimerView view;
    private TimerController controller;

    public void showDuration(String message) {
        // We could've been on the timer model's update thread!
        // So we have to return to the UI thread.
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                binding.timeTextView.setText(message);
            }
        });
    }
```

Android Activity onCreate

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    binding = ActivityMainBinding.inflate(getLayoutInflater());
    setContentView(binding.getRoot());

    showDuration("Loading...");

    TimerModel model = new TimerModel();
    view = new TimerView(model, this);
    controller = new TimerController(model);

    binding.startButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            controller.startButtonPressed();
        }
    });

    binding.stopButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            controller.stopButtonPressed();
        }
    });
}
```