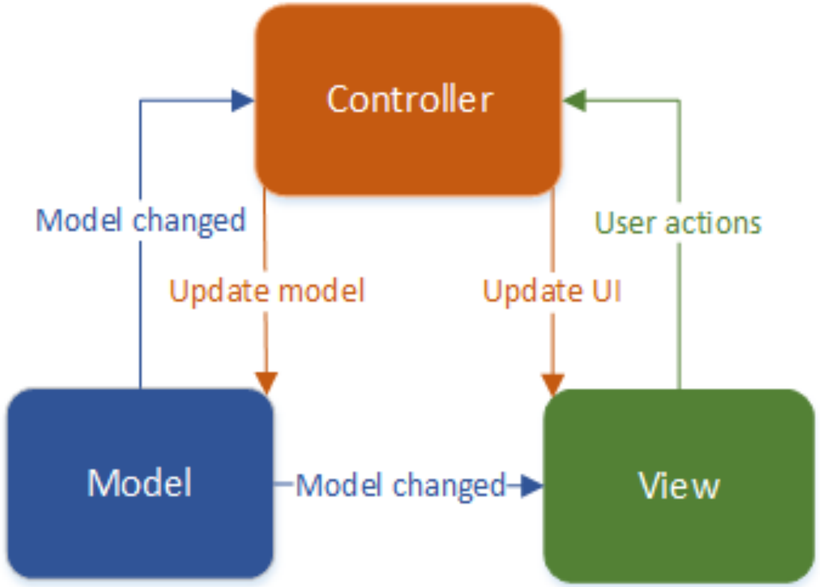
PlacemarkView / PlacemarkPresenter

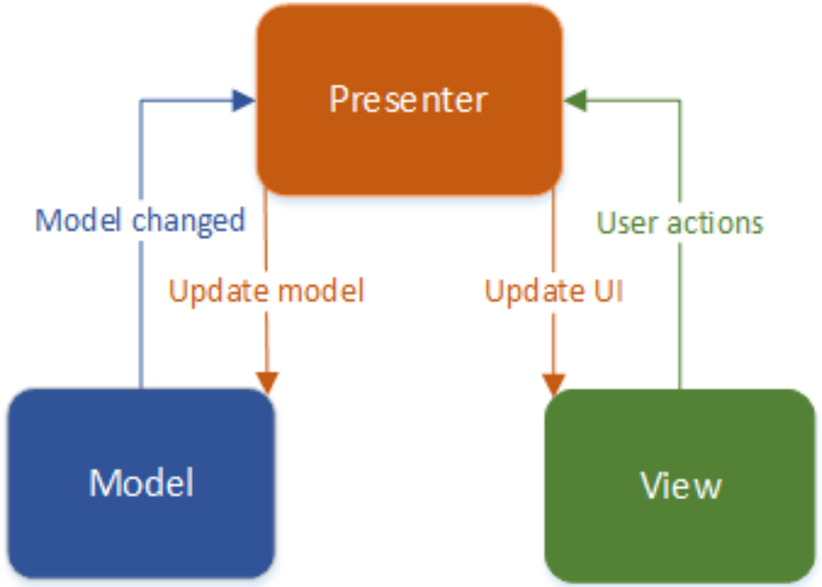
View/Presenter

Detailed review of conversion of PlacemarkActivity to PlacemarkView & PlacemarkPresenter

MVC



MVP



In MVC, the view gets notified of any change in model's state by the model itself.

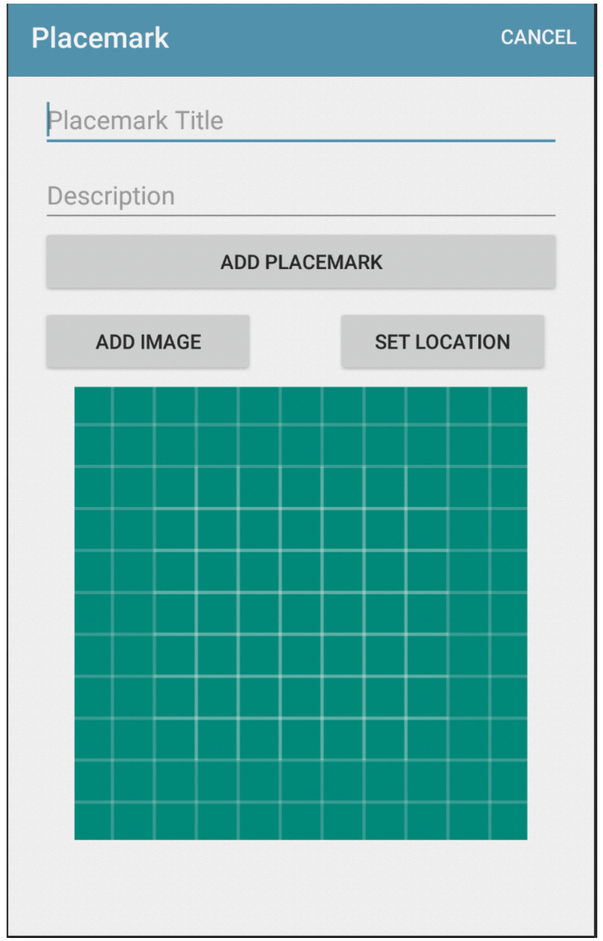
In MVP, the view knows nothing about the model, and it is the presenter's job to fetch the up to date data from the model, understand whether the view should be updated and bind a new data to the view.

Views in MVC tend to have more logic in them because they are responsible for handling of notifications from the model.

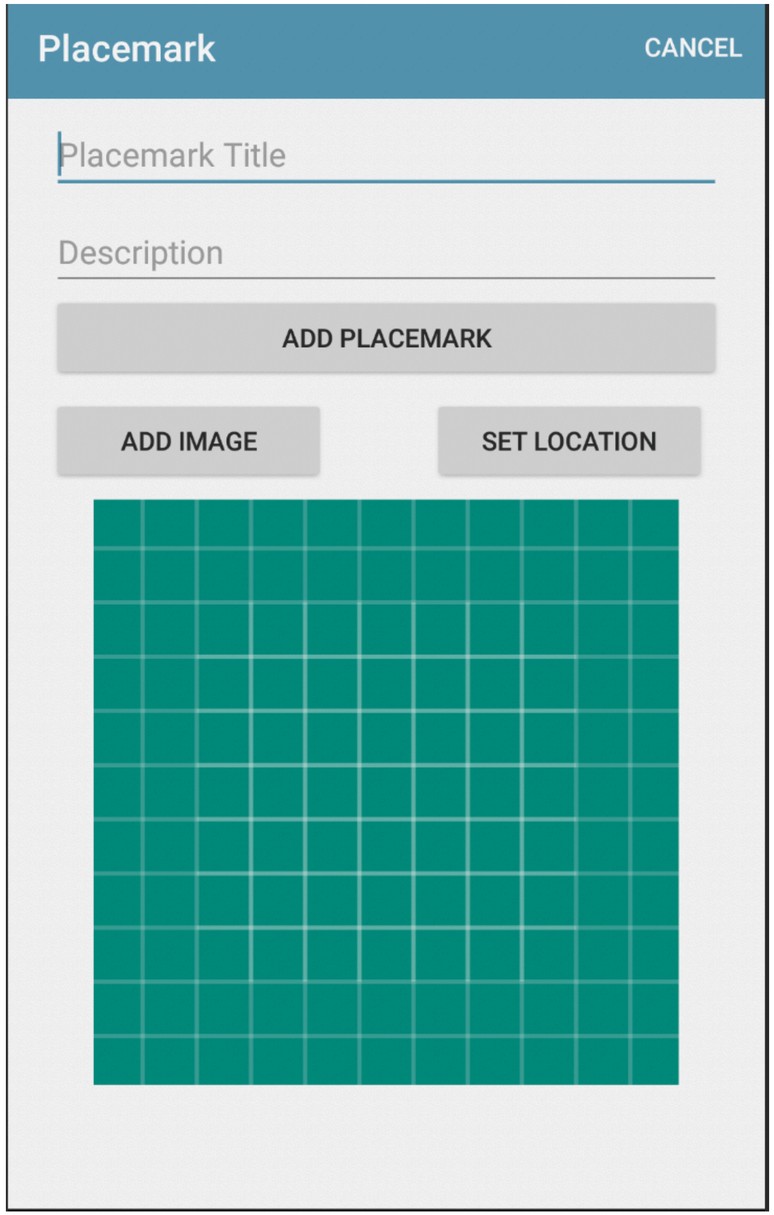
In MVP, the same logic is located in the presenter, which makes the views very "dumb" – their sole purpose becomes rendering of the data that was bound to them by the presenter and capturing user input.

```kotlin
class PlacemarkActivity : AppCompatActivity(), AnkoLogger {

  var placemark = PlacemarkModel()
  lateinit var app: MainApp
  val IMAGE_REQUEST = 1
  val LOCATION_REQUEST = 2
  var edit = false;

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_placemark)
    toolbarAdd.title = title
    setSupportActionBar(toolbarAdd)
    info("Placemark Activity started..")

    app = application as MainApp

    if (intent.hasExtra("placemark_edit")) {
      edit = true
      placemark = intent.extras.getParcelable<PlacemarkModel>("placemark_edit")
      placemarkTitle.setText(placemark.title)
      description.setText(placemark.description)
      placemarkImage.setImageBitmap(readImageFromPath(this, placemark.image))
      if (placemark.image != null) {
        chooseImage.setText(R.string.change_placemark_image)
      }
      btnAdd.setText(R.string.save_placemark)
    }

    btnAdd.setOnClickListener() {
      placemark.title = placemarkTitle.text.toString()
      placemark.description = description.text.toString()
      if (placemark.title.isEmpty()) {
        toast(R.string.enter_placemark_title)
      } else {
        if (edit) {
          app.placemarks.update(placemark.copy())
        } else {

          app.placemarks.create(placemark.copy())
        }
      }
      info("add Button Pressed: $placemarkTitle")
      setResult(AppCompatActivity.RESULT_OK)
      finish()
    }

    chooseImage.setOnClickListener {
      showImagePicker(this, IMAGE_REQUEST)
    }

    placemarkLocation.setOnClickListener {
      val location = Location(52.245696, -7.139102, 15f)
      if (placemark.zoom != 0f) {
        location.lat = placemark.lat
        location.lng = placemark.lng
        location.zoom = placemark.zoom
      }
      startActivityForResult(intentFor<MapsActivity>().putExtra("location", location), LOCATION_REQUEST)
    }
  }
```

```kotlin
  override fun onCreateOptionsMenu(menu: Menu?): Boolean {
    menuInflater.inflate(R.menu.menu_placemark, menu)
    if (edit && menu != null) menu.getItem(0).setVisible(true)
    return super.onCreateOptionsMenu(menu)
  }

  override fun onOptionsItemSelected(item: MenuItem?): Boolean {
    when (item?.itemId) {
      R.id.item_delete -> {
        app.placemarks.delete(placemark)
        finish()
      }
      R.id.item_cancel -> {
        finish()
      }
    }
    return super.onOptionsItemSelected(item)
  }

  override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    when (requestCode) {
      IMAGE_REQUEST -> {
        if (data != null) {
          placemark.image = data.getData().toString()
          placemarkImage.setImageBitmap(readImage(this, resultCode, data))
          chooseImage.setText(R.string.change_placemark_image)
        }
      }
      LOCATION_REQUEST -> {
        if (data != null) {
          val location = data.extras.getParcelable<Location>("location")
          placemark.lat = location.lat
          placemark.lng = location.lng
          placemark.zoom = location.zoom
        }
      }
    }
  }
}
```



PlacemarkActivity



PlacemarkActivity Responsibilities

- Initialising the the various controls
- Establishing the event handlers
- Overriding life cycle methods
- Interpreting menu events
- Figuring out what activities may have finished yielding results of interest
- Determining what actions to take in response to menu events
- Keeping track of edit mode
- Interacting with the model

PlacemarkActivity

- Initialising the the various controls
- Establishing the event handlers
- Overriding life cycle methods
- Interpreting menu events
- Figuring out what activities may have finished yielding results of interest
- Determining what actions to take in response to menu events
- Keeping track of edit mode
- Interacting with the model

PlacemarkView

- Initialising the the various controls
- Establishing the event handlers
- Overriding life cycle methods
- Interpreting menu events

PlacemarkPresenter

- Figuring out what activities may have finished yielding results of interest
- Determining what actions to take in response to menu events
- Keeping track of edit mode
- Interacting with the model

```kotlin
class PlacemarkView : AppCompatActivity(), AnkoLogger {

  lateinit var presenter: PlacemarkPresenter
  var placemark = PlacemarkModel()

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_placemark)
    toolbarAdd.title = title
    setSupportActionBar(toolbarAdd)

    presenter = PlacemarkPresenter(this)

    btnAdd.setOnClickListener {
      if (placemarkTitle.text.toString().isEmpty()) {
        toast(R.string.enter_placemark_title)
      } else {
        presenter.doAddOrSave(placemarkTitle.text.toString(), description.text.toString())
      }
    }

    chooseImage.setOnClickListener { presenter.doSelectImage() }

    placemarkLocation.setOnClickListener { presenter.doSetLocation() }
  }

  fun showPlacemark(placemark: PlacemarkModel) {
    placemarkTitle.setText(placemark.title)
    description.setText(placemark.description)
    placemarkImage.setImageBitmap(readImageFromPath(this, placemark.image))
    if (placemark.image != null) {
      chooseImage.setText(R.string.change_placemark_image)
    }
    btnAdd.setText(R.string.save_placemark)
  }

  override fun onCreateOptionsMenu(menu: Menu): Boolean {
    menuInflater.inflate(R.menu.menu_placemark, menu)
    if (presenter.edit) menu.getItem(0).setVisible(true)
    return super.onCreateOptionsMenu(menu)
  }

  override fun onOptionsItemSelected(item: MenuItem?): Boolean {
    when (item?.itemId) {
      R.id.item_delete -> {
        presenter.doDelete()
      }
      R.id.item_cancel -> {
        presenter.doCancel()
      }
    }
    return super.onOptionsItemSelected(item)
  }

  override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    if (data != null) {
      presenter.doActivityResult(requestCode, resultCode, data)
    }
  }
```

- Initialising the the various controls
- Establishing the event handlers
- Overriding life cycle methods
- Interpreting menu events

PlacemarkView

- Initialising the the various controls
- Establishing the event handlers

```
class PlacemarkView : AppCompatActivity(), AnkoLogger {

  lateinit var presenter: PlacemarkPresenter
  var placemark = PlacemarkModel()

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_placemark)
    toolbarAdd.title = title
    setSupportActionBar(toolbarAdd)

    presenter = PlacemarkPresenter(this)

    btnAdd.setOnClickListener {
      if (placemarkTitle.text.toString().isEmpty()) {
        toast(R.string.enter_placemark_title)
      } else {
        presenter.doAddOrSave(placemarkTitle.text.toString(), description.text.toString())
      }
    }

    chooseImage.setOnClickListener { presenter.doSelectImage() }

    placemarkLocation.setOnClickListener { presenter.doSetLocation() }
  }
}
```

Create the **Presenter** object

Defer all decisions to the **Presenter** object

PlacemarkView

```
class PlacemarkView : AppCompatActivity(), AnkoLogger {

  lateinit var presenter: PlacemarkPresenter
  var placemark = PlacemarkModel()

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_placemark)
    toolbarAdd.title = title
    setSupportActionBar(toolbarAdd)

    presenter = PlacemarkPresenter(this)

    btnAdd.setOnClickListener {
      if (placemarkTitle.text.toString().isEmpty()) {
        toast(R.string.enter_placemark_title)
      } else {
        presenter.doAddOrSave(placemarkTitle.text.toString(), description.text.toString())
      }
    }

    chooseImage.setOnClickListener { presenter.doSelectImage() }

    placemarkLocation.setOnClickListener { presenter.doSetLocation() }
  }
}
```

PlacemarkView

- Overriding life cycle methods
- Interpreting menu events

```
class PlacemarkView : AppCompatActivity(), AnkoLogger {

  lateinit var presenter: PlacemarkPresenter
  var placemark = PlacemarkModel()
 ...

  override fun onOptionsItemSelected(item: MenuItem?): Boolean {
    when (item?.itemId) {
      R.id.item_delete -> {
        presenter.doDelete()
      }
      R.id.item_cancel -> {
        presenter.doCancel()
      }
    }
    return super.onOptionsItemSelected(item)
  }

  override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    if (data != null) {
      presenter.doActivityResult(requestCode, resultCode, data)
    }
  }
}
```

Defer all decisions to the **Presenter** object

PlacemarkView

```
class PlacemarkView : AppCompatActivity(), AnkoLogger {

  lateinit var presenter: PlacemarkPresenter
  var placemark = PlacemarkModel()
 ...

  override fun onOptionsItemSelected(item: MenuItem?): Boolean {
    when (item?.itemId) {
      R.id.item_delete -> {
        presenter.doDelete()
      }
      R.id.item_cancel -> {
        presenter.doCancel()
      }
    }
    return super.onOptionsItemSelected(item)
  }

  override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    if (data != null) {
      presenter.doActivityResult(requestCode, resultCode, data)
    }
  }
}
```

PlacemarkView

PlacemarkPresenter

```
class PlacemarkPresenter(val view: PlacemarkView) {

  val IMAGE_REQUEST = 1
  val LOCATION_REQUEST = 2

  var placemark = PlacemarkModel()
  var location = Location(52.245696, -7.139102, 15f)
  var app: MainApp
  var edit = false;

  init {
    app = view.application as MainApp
    if (view.intent.hasExtra("placemark_edit")) {
      edit = true
      placemark = view.intent.extras.getParcelable<PlacemarkModel>("placemark_edit")
      view.showPlacemark(placemark)
    }
  }

  fun doAddOrSave(title: String, description: String) {
    placemark.title = title
    placemark.description = description
    if (edit) {
      app.placemarks.update(placemark)
    } else {
      app.placemarks.create(placemark)
    }
    view.finish()
  }

  fun doCancel() {
    view.finish()
  }

  fun doDelete() {
    app.placemarks.delete(placemark)
    view.finish()
  }

  fun doSelectImage() {
    showImagePicker(view, IMAGE_REQUEST)
  }

  fun doSetLocation() {
    if (placemark.zoom != 0f) {
      location.lat = placemark.lat
      location.lng = placemark.lng
      location.zoom = placemark.zoom
    }
    view.startActivityForResult(view.intentFor<EditLocationView>().putExtra("location", location), LOCATION_REQUEST)
  }

  fun doActivityResult(requestCode: Int, resultCode: Int, data: Intent) {
    when (requestCode) {
      IMAGE_REQUEST -> {
        placemark.image = data.data.toString()
        view.showPlacemark(placemark)
      }
      LOCATION_REQUEST -> {
        location = data.extras.getParcelable<Location>("location")
        placemark.lat = location.lat
        placemark.lng = location.lng
        placemark.zoom = location.zoom
      }
    }
  }
}
```

- Determining what actions to take in response to menu events
- Keeping track of edit mode
- Interacting with the model
- Figuring out what activities may have finished yielding results of interest

PlacemarkPresenter

Accept the View in the constructor

```
class PlacemarkPresenter(val view: PlacemarkView) {

  val IMAGE_REQUEST = 1
  val LOCATION_REQUEST = 2

  var placemark = PlacemarkModel()
  var location = Location(52.245696, -7.139102, 15f)
  var app: MainApp
  var edit = false;

  init {
    app = view.application as MainApp
    if (view.intent.hasExtra("placemark_edit")) {
      edit = true
      placemark = view.intent.extras.getParcelable<PlacemarkModel>("placemark_edit")
      view.showPlacemark(placemark)
    }
  }
```

Ask view to display the place mark

Each method prefixed with **do**

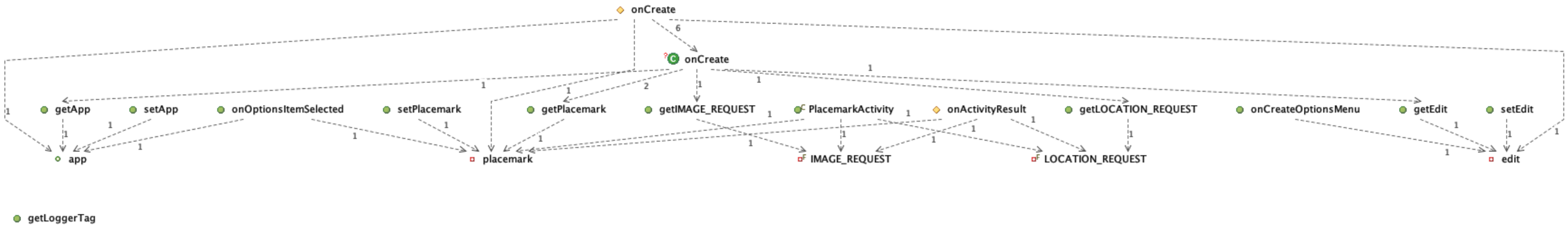
These methods invoked by View in response to user interaction

Update the model if necessary

If the view is to be updated, invoke view methods

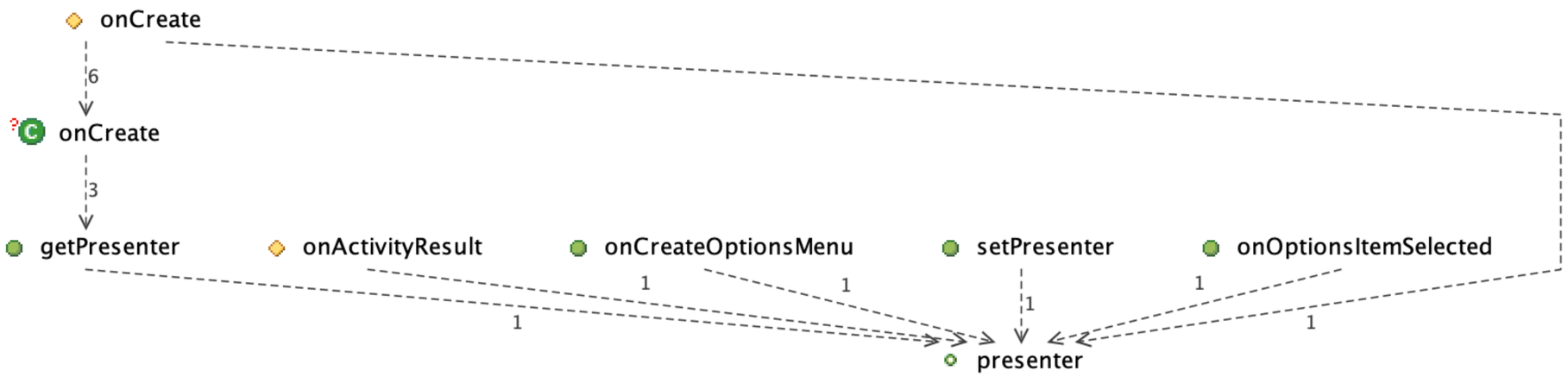
PlacemarkPresenter

```
fun doAddOrSave(title: String, description: String) {
  placemark.title = title
  placemark.description = description
  if (edit) {
    app.placemarks.update(placemark)
  } else {
    app.placemarks.create(placemark)
  }
  view.finish()
}

fun doCancel() {
  view.finish()
}

fun doDelete() {
  app.placemarks.delete(placemark)
  view.finish()
}

fun doSelectImage() {
  showImagePicker(view, IMAGE_REQUEST)
}

fun doSetLocation() {
  if (placemark.zoom != 0f) {
    location.lat = placemark.lat
    location.lng = placemark.lng
    location.zoom = placemark.zoom
  }
  view.startActivityForResult(view.intentFor<EditLocationView>().putExtra("location", location), LOCATI
}

fun doActivityResult(requestCode: Int, resultCode: Int, data: Intent) {
  when (requestCode) {
    IMAGE_REQUEST -> {
      placemark.image = data.data.toString()
      view.showPlacemark(placemark)
    }
    LOCATION_REQUEST -> {
      location = data.extras.getParcelable<Location>("location")
      placemark.lat = location.lat
      placemark.lng = location.lng
      placemark.zoom = location.zoom
    }
  }
}
```



PlacemarkActivity Complexity

PlacemarkView

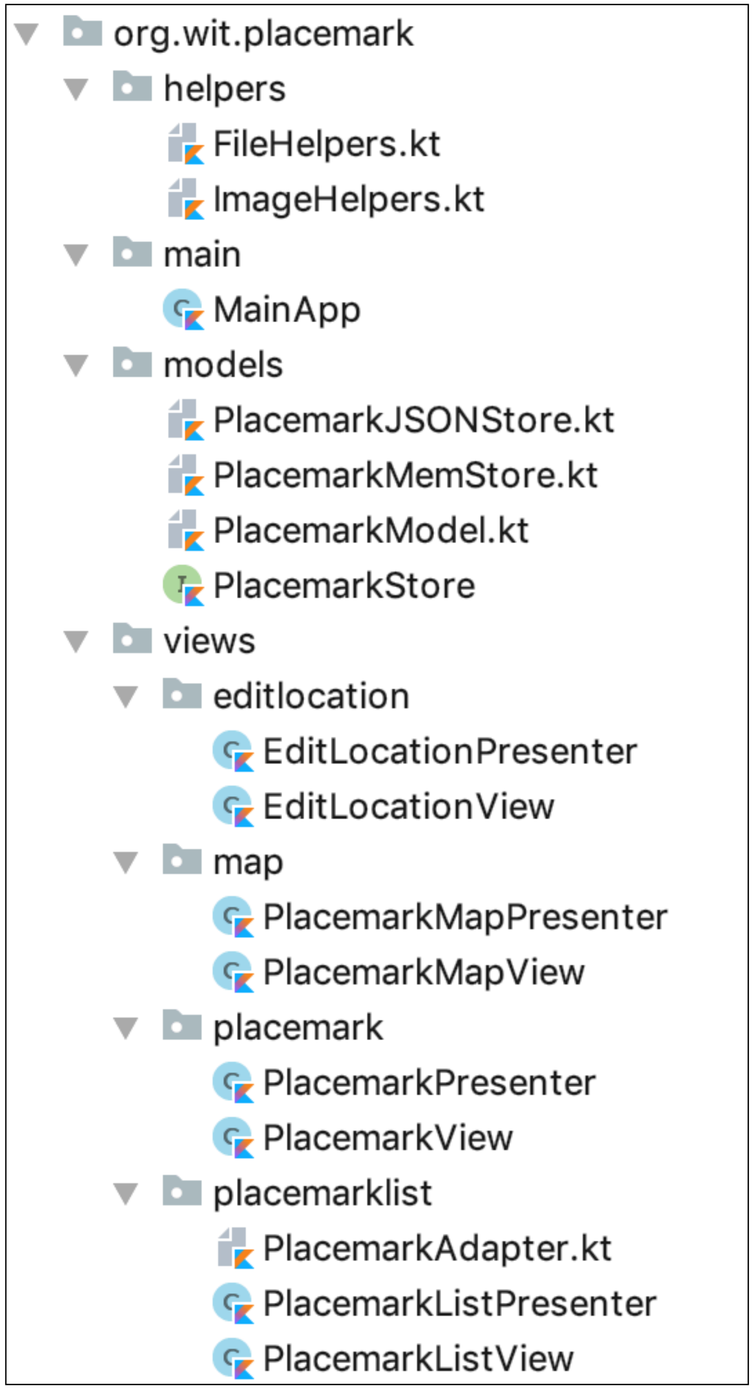


PlacemarkPresenter

- org.wit.placemark
  - activities
    - MapsActivity
    - PlacemarkActivity
    - PlacemarkAdapter.kt
    - PlacemarkListActivity
    - PlacemarkMapsActivity
  - helpers
    - FileHelpers.kt
    - ImageHelpers.kt
  - main
    - MainApp
  - models
    - PlacemarkJSONStore.kt
    - PlacemarkMemStore.kt
    - PlacemarkModel.kt
    - PlacemarkStore

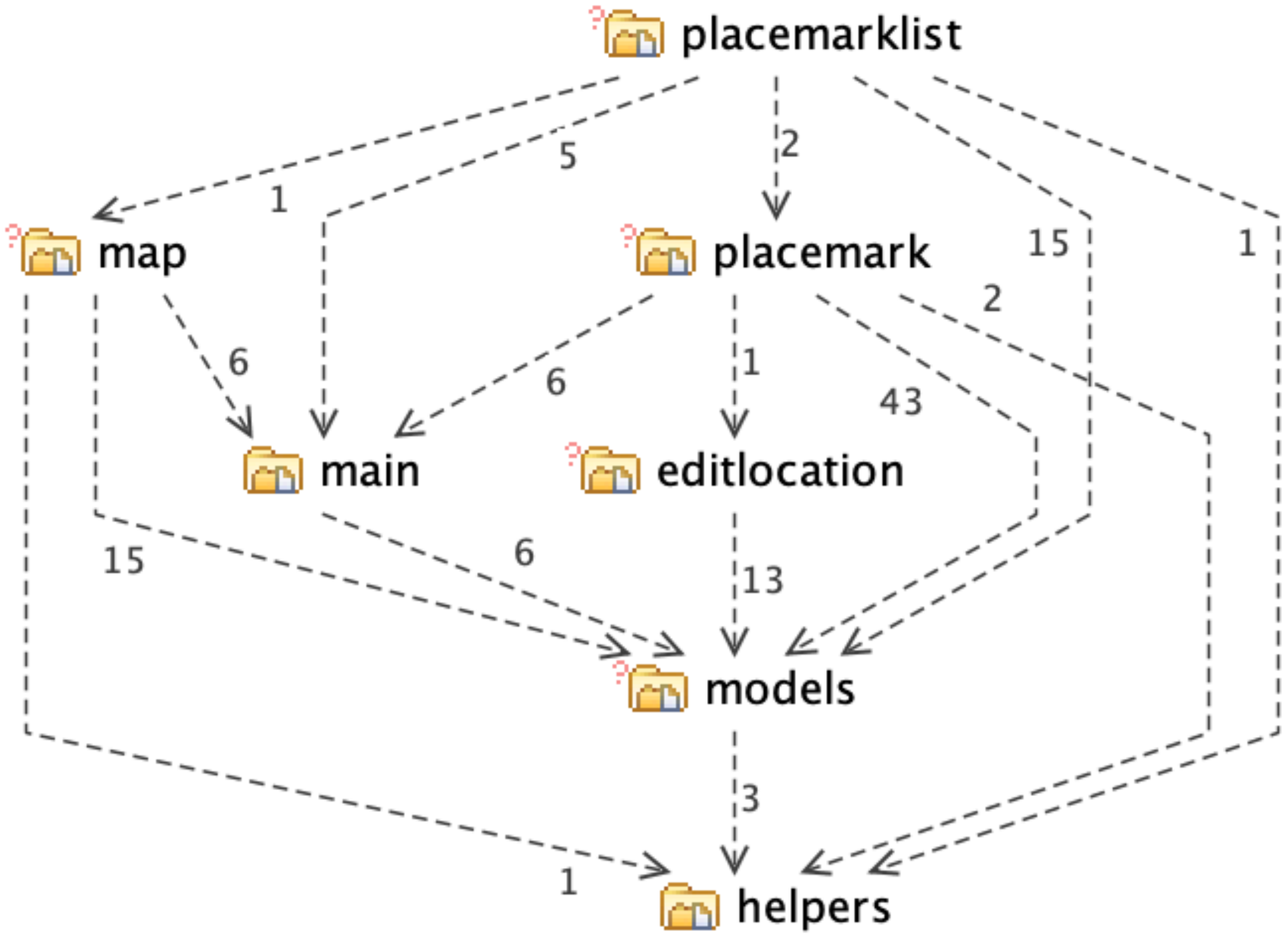
Refactor to MVP

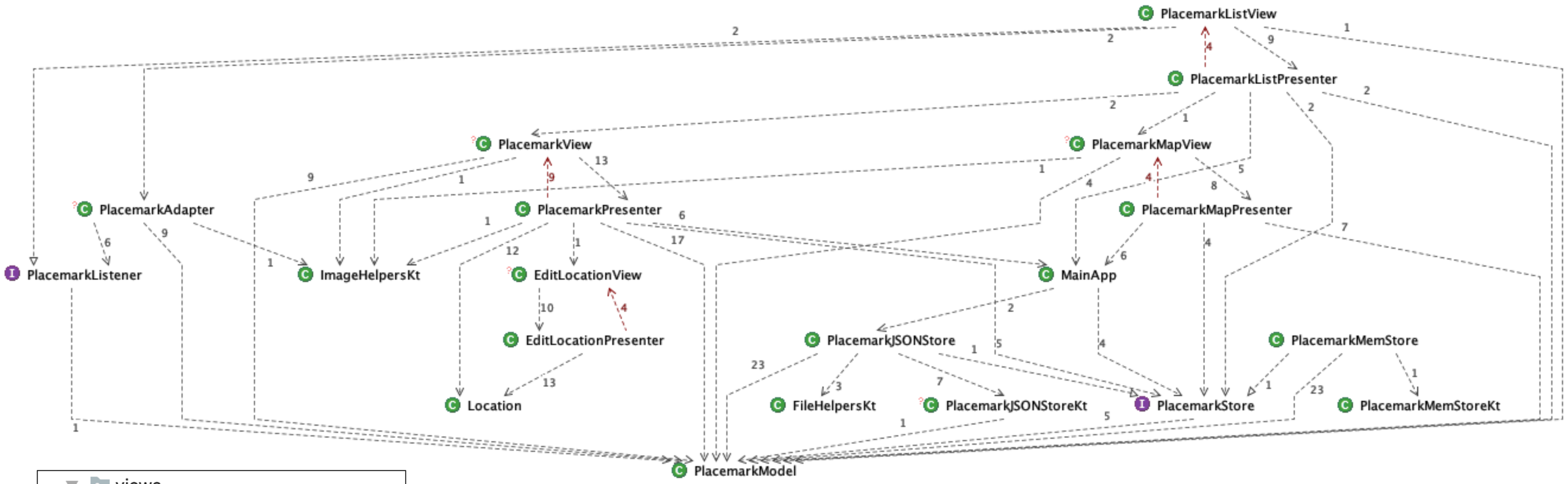
- org.wit.placemark
  - helpers
    - FileHelpers.kt
    - ImageHelpers.kt
  - main
    - MainApp
  - models
    - PlacemarkJSONStore.kt
    - PlacemarkMemStore.kt
    - PlacemarkModel.kt
    - PlacemarkStore
  - views
    - editlocation
      - EditLocationPresenter
      - EditLocationView
    - map
      - PlacemarkMapPresenter
      - PlacemarkMapView
    - placemark
      - PlacemarkPresenter
      - PlacemarkView
    - placemarklist
      - PlacemarkAdapter.kt
      - PlacemarkListPresenter
      - PlacemarkListView



- 4 Activities supported by application - *placemark, placemarklist, editlocation, map*
- Refactor all Presenter/Views into new **views** package
- Each application activity delivered by Presenter/View classes in a package

- org.wit.placemark
  - helpers
    - FileHelpers.kt
    - ImageHelpers.kt
  - main
    - MainApp
  - models
    - PlacemarkJSONStore.kt
    - PlacemarkMemStore.kt
    - PlacemarkModel.kt
    - PlacemarkStore
  - views
    - editlocation
      - EditLocationPresenter
      - EditLocationView
    - map
      - PlacemarkMapPresenter
      - PlacemarkMapView
    - placemark
      - PlacemarkPresenter
      - PlacemarkView
    - placemarklist
      - PlacemarkAdapter.kt
      - PlacemarkListPresenter
      - PlacemarkListView





- views
  - editlocation
    - EditLocationPresenter
    - EditLocationView
  - map
    - PlacemarkMapPresenter
    - PlacemarkMapView
  - placemark
    - PlacemarkPresenter
    - PlacemarkView
  - placemarklist
    - PlacemarkAdapter.kt
    - PlacemarkListPresenter
    - PlacemarkListView

- helpers
  - FileHelpers.kt
  - ImageHelpers.kt
- main
  - MainApp
- models
  - PlacemarkJSONStore.kt
  - PlacemarkMemStore.kt
  - PlacemarkModel.kt
  - PlacemarkStore