Presenter First: Organizing complex GUI applications for TDD

Carl Erickson
Atomic Object LLC
GUI Test Automation

• Lesson from Haste
  – GUI test automation is hard
  – expensive to build
  – difficult to make completely deterministic
  – expensive to run (GUI widget creation)
  – not really suitable for SDD

• Thin GUI
  – Trust the GUI objects are already tested
  – Confirm visually the view’s composition
  – Allow no logic in the view
Model View Presenter (MVP)

- Goes way back (Dolphin Smalltalk, C++ framework)
- Variation on classic MVC
- A key pattern for doing Presenter First
  - But by no means the whole picture
PF Variant on MVP

Original MVP

Presenter First

Dashed lines are events, solid are object messaging.
Where to Start?

• Model
  – infrastructure-first temptation
  – readily testable, can use TDD

• View
  – better, as it’s closer to the user
  – hard to do TDD (Haste lesson)
  – risk of fattening the view with business logic
  – likely to change, attracts strong opinions
Presenter First

• Start with the presenter
  – TDD works well (no interface)
  – Presenter unit tests correspond to user stories
• Model and View interfaces
  – Specifications for Model and View discovered
  – Easily mocked for testing
Presenter

• Intentionally stateless
  – Wires the app together according to features, but holds no state and computes little or nothing

• No public API
  – Communication with the Presenter is via events (looser coupling)
Create a stub presenter class that takes a model interface and a view interface in its constructor
Create mock test objects that satisfy the model and view interfaces
For all user stories
1. Analyze a prioritized user story for impact on view
2. Add support to view interface for the story
3. Analyze story for impact on model
4. Add support to model interface for the story
5. Implement methods in the mock objects for the new interface methods which confirm they were called, satisfy return type
6. For all things that can break (TDD loop)
   Do for each test
       Write a test for the presenter that exercises the app via an event or action on the view or model (an external system event)
       Make assertions on the state of the model and the state of the view
       Implement private methods in presenter
   Until the test passes
7. Create a minimal user interface implementation
Use of Events

• View & Model fire events consumed by Presenter
  – looser coupling than messaging
  – separate packaging, no re-compile
  – supports multiple views (or models)
  – view is very shallow (click and fire), has no behavior of its own
  – view takes only primitive types (no processing)
Importance of Tools

• Creating MVP triads
  – There are a lot of them
  – Each has interfaces, mocks and concrete classes

• Navigating MVP triads
  – Moving between classes readily

• Composing an application from MVP triads
  – All from main() - scalability issue
  – Declaratively via DI framework (Spring)
Complex View Widgets

• Comlex, domain-specific GUI widgets
  – Requires non-primitive params, hence logic, hence testing

• Polluting the view or the model
  – Yuck

• Use of adaptors
  – Translate GUI-centric events & data to domain-centric terms
  – Keeps the view and model clean
MVP Adaptors
Example: Dyno Controller

• Dynamometer Controller
  – Drag/drop programming of tests
  – Sophisticated GUI interaction
  – Presentation of app-specific data structures

• Code bulk
  – 1200 classes / 490 interfaces
  – 100 models, 120 views, 100 presenters
  – 22 adaptors

• Test bulk
  – 5000+ tests
Java Example

Authentication challenge (login screen)

- **Model**
  - Business logic of validating passwords
- **View**
  - Prompts for password
  - Displays “success” or “denied” message
- **Presenter**
  - Represents the feature of challenge, response
Benefits

• Concentrate on user-defined behavior, not implementation details (model and view)
• Interfaces/events from M, V, P eases testing
• Fast running tests with mock view (no widgets)
• Division of labor, skills, teams
• Full application system (almost) testing