

Hard Things to Test and Refactor

Legacy Code

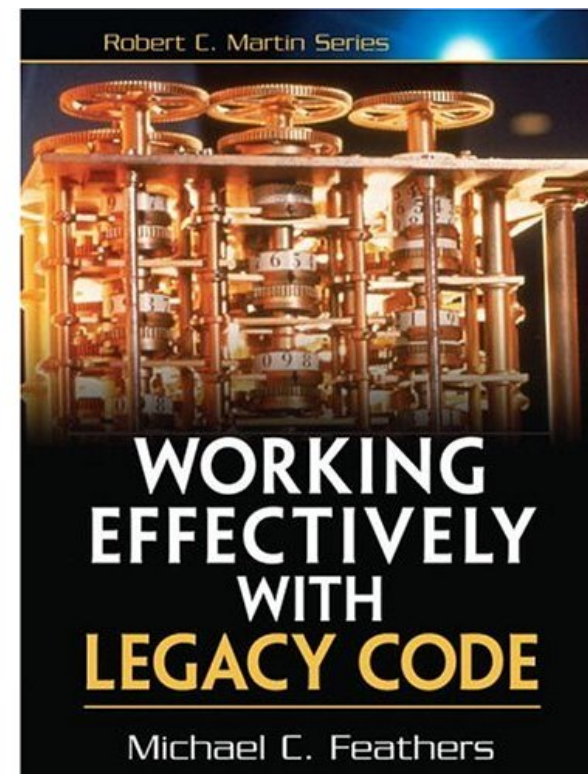
- Using good development techniques is easy in greenfield projects, but most projects carry some amount of legacy code, even 100 or 1000 times more than new code.
- *Legacy code* is code without tests.

” A few years ago, I asked a friend how his new client was doing. He said "they're writing legacy code." [...] The age of the code has nothing to do with it. People are writing legacy code right now, maybe on your project.

” Most of the fear involved in making changes to large code bases is fear of introducing subtle bugs; fear of changing things inadvertently. With tests, you can make things better with impunity.

Dealing With Legacy Code

- Create "test coverings" to introduce an invariant over a small area of the system.
 - Know when we have changed the behaviour of the system.
 - Correct behaviour is defined by what the system did yesterday.
- Start from the parts which need to be changed first. Slowly expand the test coverings to cover the whole system.



Dealing With Legacy Code

- In general, tests (1) seed the design, (2) record intentions of its designers, (3) act as an invariant on the code.
 - With legacy code we produce the most value by working backwards: build the invariant first, then refactor to make the code clean and add new behaviour.
- Systems that are in production need more diligent covering, as users have come to depend upon the current behaviour.
- Systems that have not yet been deployed can be refactored with relative impunity – no one knows anyways whether it worked yet.

General Legacy Management Strategy

1. Identify change points
2. Find an inflection point
3. Cover the inflection point
 - a) Break external dependencies
 - b) Break internal dependencies
 - c) Write tests
4. Make changes
5. Refactor the covered code

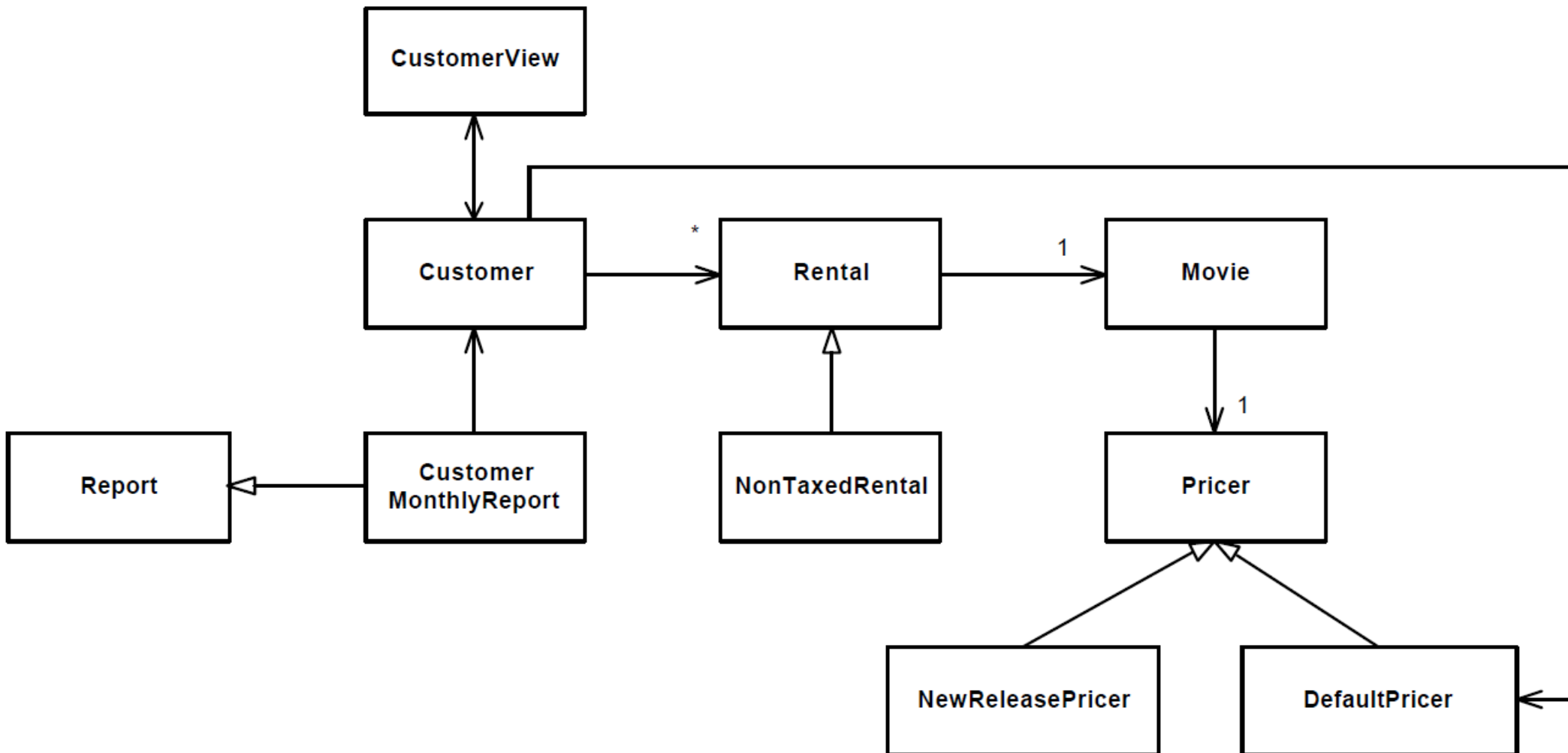
1. Identify Change Points

- First figure out where the changes will need to be made.
- If there are many ways to make the changes, and there are not yet test coverings in place, choose the way that requires the fewest changes.
 - Saves time. Covering large areas of code with tests is slow.
- When there are more test coverings in place, the classes can be refactored more to make the design right.

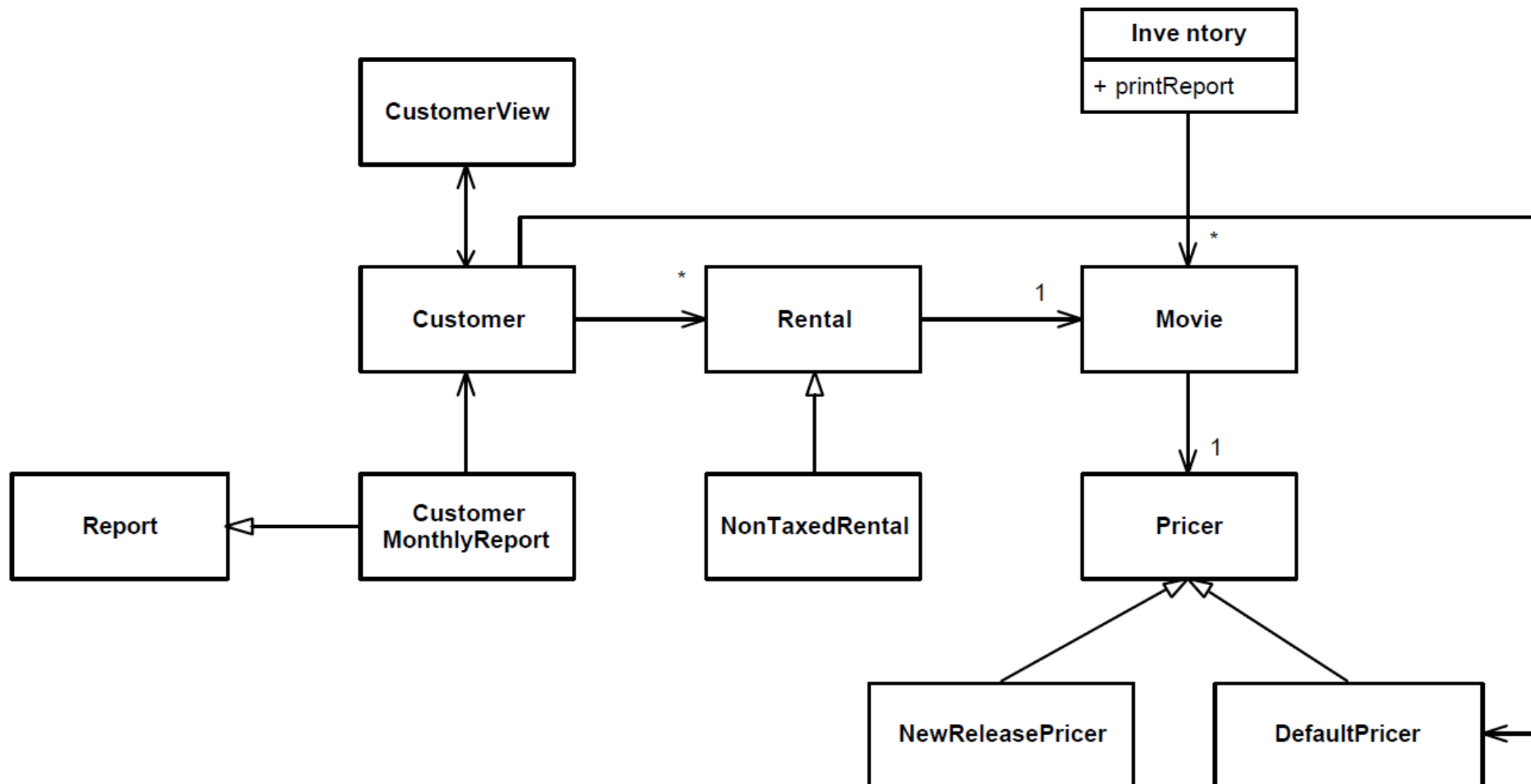
2. Find an Inflection Point

- An *inflection point* is a narrow interface to a set of classes. If anyone changes any of the classes behind an inflection point, the change is either detectable at the inflection point, or inconsequential in the application.
- Do not consider only physical dependencies, but also the way that effects are propagated at runtime.
- When looking for inflection points, move outward from the places you are going to change. Look for a narrow interface. It could be one class or several.
- Do not trust existing UML diagrams. They rarely show all users of a class, and they can be out of date.

Customer is the inflection point for Rental, Movie, Pricer and their subclasses.



Customer is still the inflection point for Rental, but changes to Movie and Pricer might propagate to the rest of the system through Inventory.



3. Cover the Inflection Point

- Covering an inflection point involves writing a tests for it. The hard part of this is getting your legacy code to compile in a test harness. You often have to break dependencies.
- Dependencies of a class can often be found by just trying to create a new instance of the class.
- There are two types of dependencies:
 - *External dependencies* are objects which we have to provide to setup the object we are creating.
 - *Internal dependencies* are objects which the class creates by itself.

3a. Breaking External Dependencies

- Objects talk to other objects. To test an object, we need to sever the connection between it and other objects. This can be done with the *Dependency Inversion Principle*:
 - A) High level modules should not depend upon low level modules. Both should depend upon abstractions.
 - B) Abstractions should not depend upon details. Details should depend upon abstractions.

```
class CustomerView {
    private Customer _customer;
    public void setCustomer(Customer customer) {
        _customer = customer;
    }
    public void update() {
        nameWidget.setText(_customer.getName());
    }
    ...
}
```

```
class Customer {
    public Customer(CustomerView view) {
        _view = view;
        _view.setCustomer(this);
    }
    ...
}
```



Depends on CustomerView
and everything that
it depends on.

```
interface CustomerView {
    void setCustomer(Customer customer);
    void update();
}
```

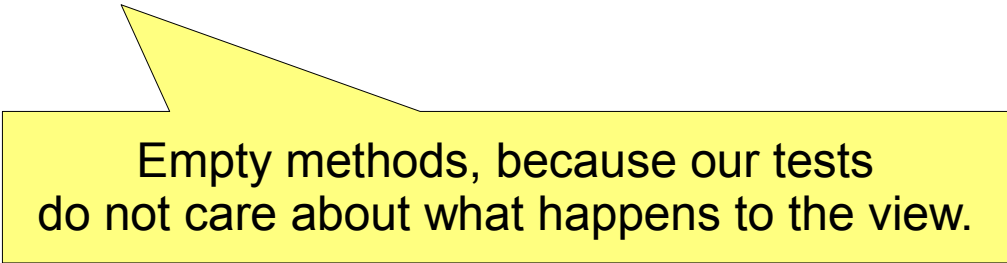
```
class StandardCustomerView implements CustomerView {
    private Customer _customer;
    public void setCustomer(Customer customer) {
        _customer = customer;
    }
    public void update() {
        nameWidget.setText(_customer.getName());
    }
    ...
}
```

```
class Customer {
    public Customer(CustomerView view) {
        _view = view;
        _view.setCustomer(this);
    }
    ...
}
```



Depends only on
an interface.

```
Customer customer = new Customer(new CustomerView () {  
    public void setCustomer(Customer customer) {}  
    public void update() {}  
});
```

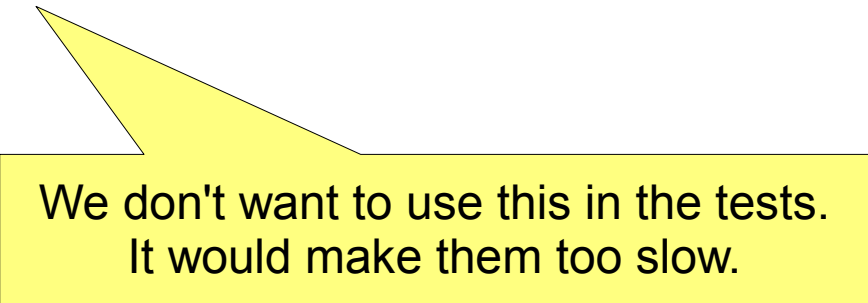


Empty methods, because our tests do not care about what happens to the view.

3b. Breaking Internal Dependencies

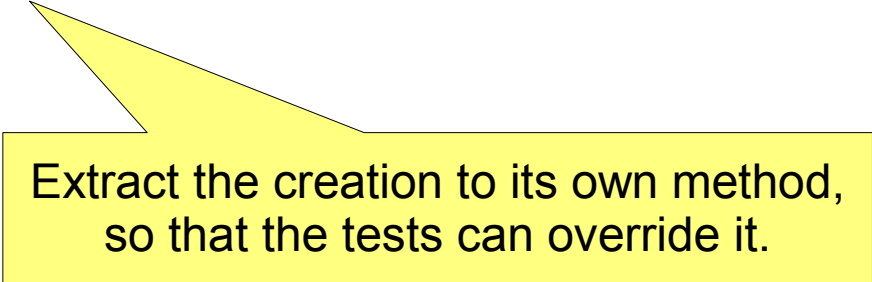
- When the class we want to cover creates its own objects internally, sometimes the best thing that you can do is subclass to override the creations.
- Global variables are also internal dependencies. In an OO system they appear as Singletons or static data classes.
 - You must provide them with a good known initial state. If you forget something, the side-effects easily bleed from one test to other tests.
 - Singletons which do not affect the functional behavior of an application (caches, factories), can behave well as internal dependencies.

```
class Customer {
    private Archiver _archiver;
    public Customer(CustomerView view) {
        ...
        archiver = new FileArchiver(customerPersistenceName);
        ...
    }
}
```



We don't want to use this in the tests.
It would make them too slow.


```
class Customer {
    private Archiver archiver;
    public Customer(CustomerView view) {
        ...
        archiver = makeArchiver();
        ...
    }
    protected Archiver makeArchiver() {
        return new FileArchiver(customerPersistenceName);
    }
}
```



Extract the creation to its own method,
so that the tests can override it.

```
class TestingCustomer extends Customer {  
    protected Archiver makeArchiver() {  
        return new NullArchiver();  
    }  
}
```

Null Object Pattern

```
Customer customer = new TestingCustomer(  
    new CustomerView() {  
        public void setCustomer(Customer customer) {}  
        public void update() {}  
    }  
);
```

Null Object Pattern

```
Customer customer =  
    new TestingCustomer(new NullCustomerView());
```

3c. Writing Tests

- Once the objects of an inflection point can be created in a test, we need to place some sort of an invariant on the code guarded by the inflection point.
- Goal: Code changes behind the inflection point can not have effect in the system without passing through the inflection point.
 - Write tests for the interfaces of the inflection point.
 - Correctness is defined by what the system does currently.
 - Try equivalence partitions and boundary values.
 - Try changing the system and see if the tests notice it.
 - Automated test generation may also be possible.
 - JUnit Factory (<http://www.agitar.com/>)

4-5. Make Changes and Refactor

- Run tests often when making changes. Write more tests for the changes you make, to improve the test suite.
- Clean up the code with refactorings:
 - Extract method over and over again.
 - Pay attention to groups of methods which *use* other methods and data in the class. If there are any, there might be a good place to split the class with the extract class refactoring.
 - Remember to write more tests as you refactor. Even though there are test coverings, they might not test what you think they do.
 - *Refactoring into tests:* In the case of extract method, you look at a large method, imagine a portion you'd like to extract, write a test for it, then you extract the method to make the test pass.

GUI Testing

- Test-driving user interfaces is challenging.
 - *Brittle tests*: If the tests become coupled to the presentation details, changing look and feel of the UI breaks tests.
 - *Slow tests*: UI frameworks are generally complicated and may slow down the tests much. Even a full web server and browser might be needed to test a web UI.
 - *Nonformal specification*: Writing a test which defines that the UI looks and works right is practically impossible. Verification by a human is the only possibility.
- The general solution is to minimize the UI code that can not be tested automatically. Separate the UI logic from its presentation.

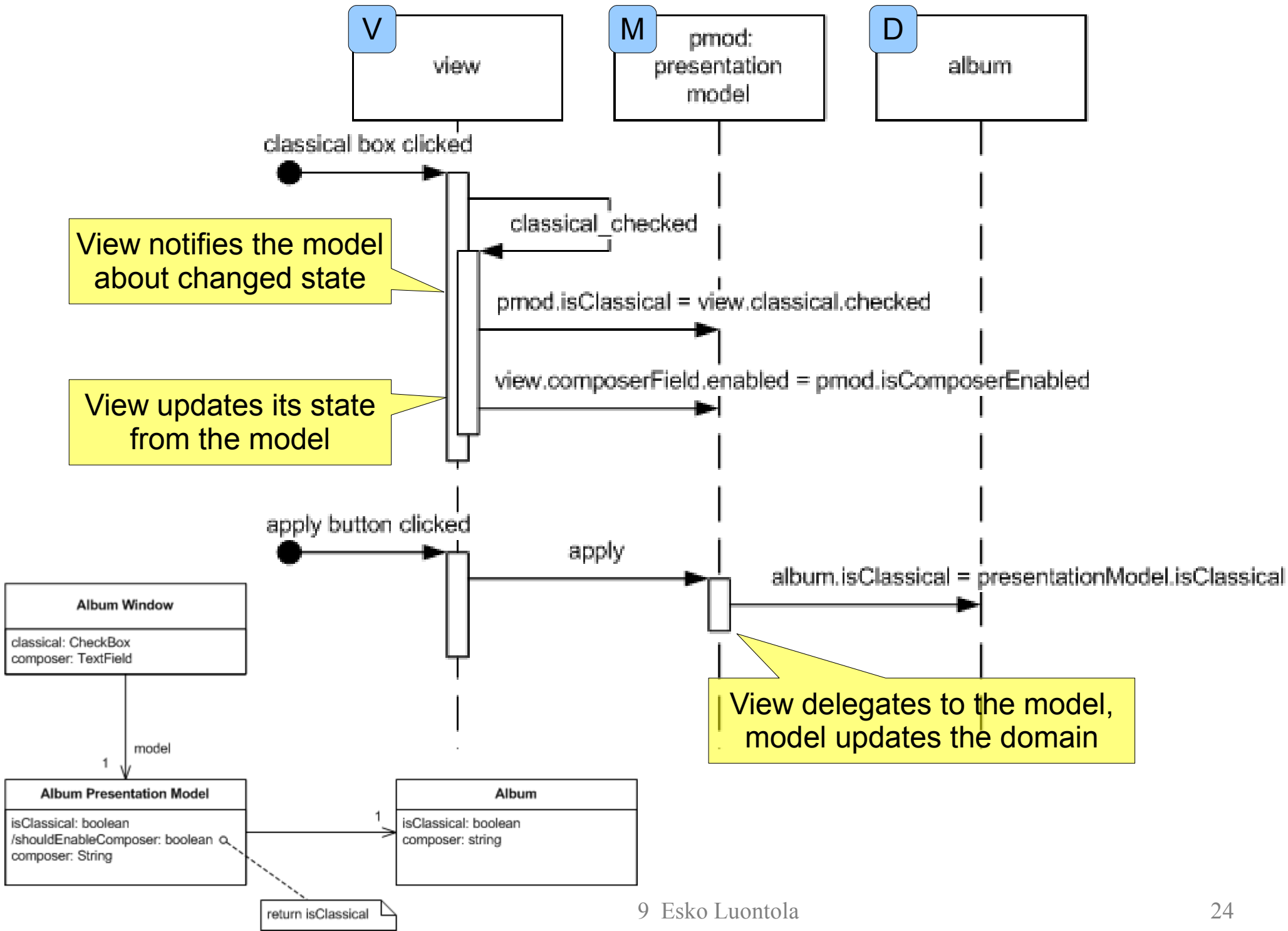
Declarative Data Rarely Breaks

- When test-driving UI's it is important to decide what to test and what **not** to test. The general rule for TDD is to *test anything that can break*. The corollary, especially relevant for UI's, is *don't test anything when you don't care if it changes*. It is usually the *dynamic behavior* of the UI that can break and should be tested.
- In general, declarative data does not need to be tested:
 - If a program reads a username and password from a config file, test the *dynamic behaviour* of reading them. Do not test the *declarative data* that are the username and password values correct.
 - Otherwise the tests would just duplicate the production code. If you need to change one, you also have to change the other.
 - See also: different Tetromino shapes.

Presentation Model

Represent the state and behavior of the presentation independently of the GUI controls used in the interface

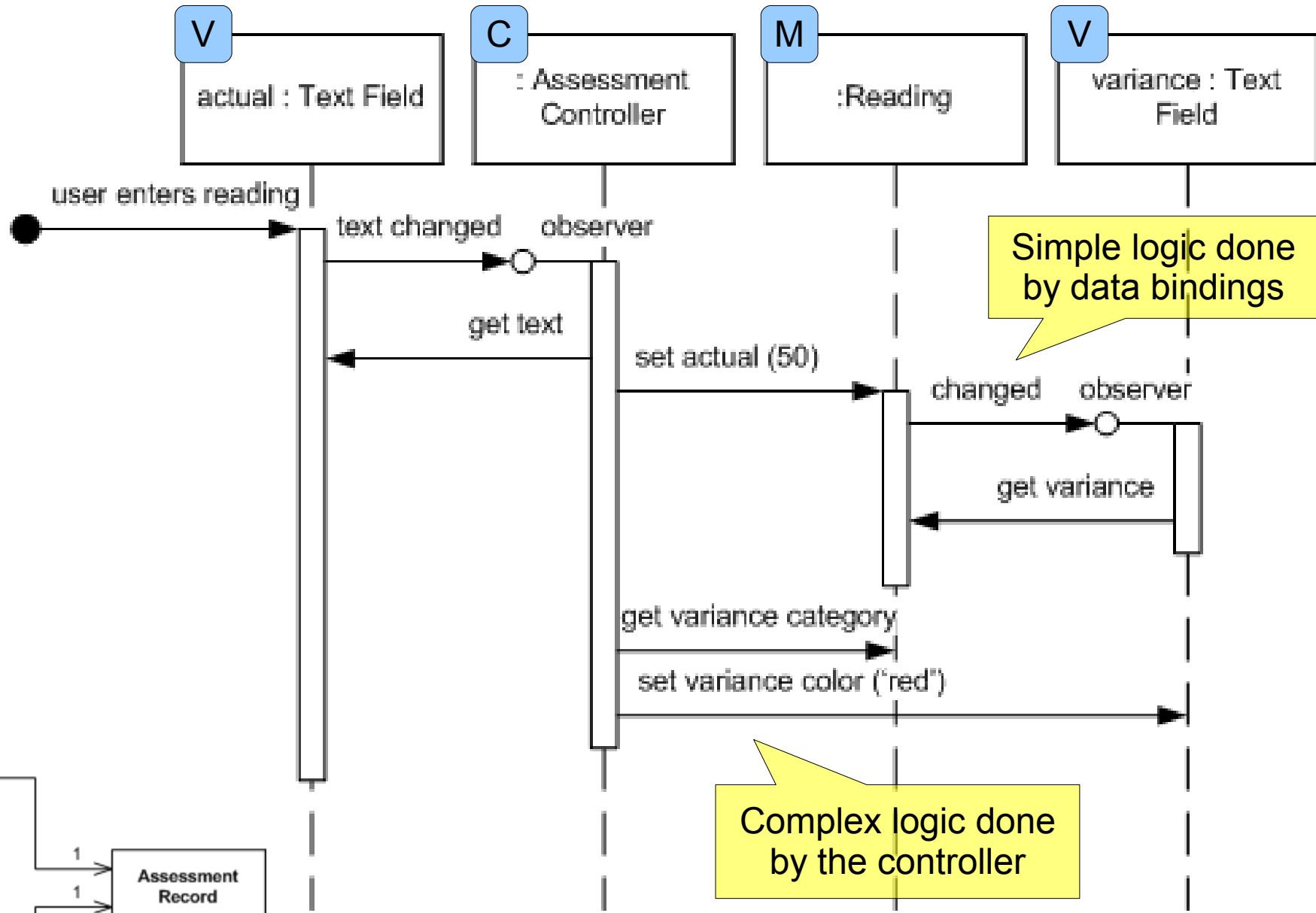
- Presentation Model is of a fully self-contained class that represents all the data and behavior of the UI window, but without any of the controls used to render that UI on the screen. A view then simply projects the state of the presentation model onto the glass. Each view should require only one Presentation Model.
- To do this the Presentation Model will have data fields for all the dynamic information of the view. This won't just include the contents of controls, but also things like whether or not they are enabled.
- Testing the synchronization code can be hard.



Supervising Controller

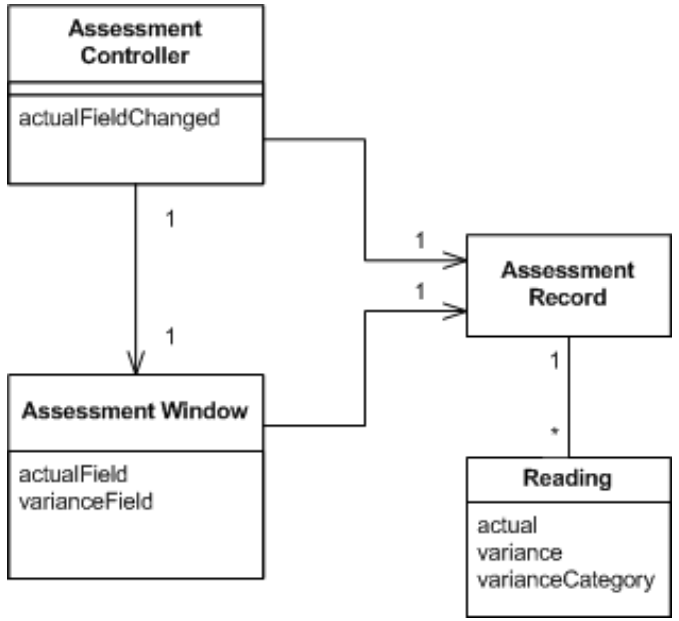
Factor the UI into a view and controller where the view handles simple mapping to the underlying model and the the controller handles input response and complex view logic.

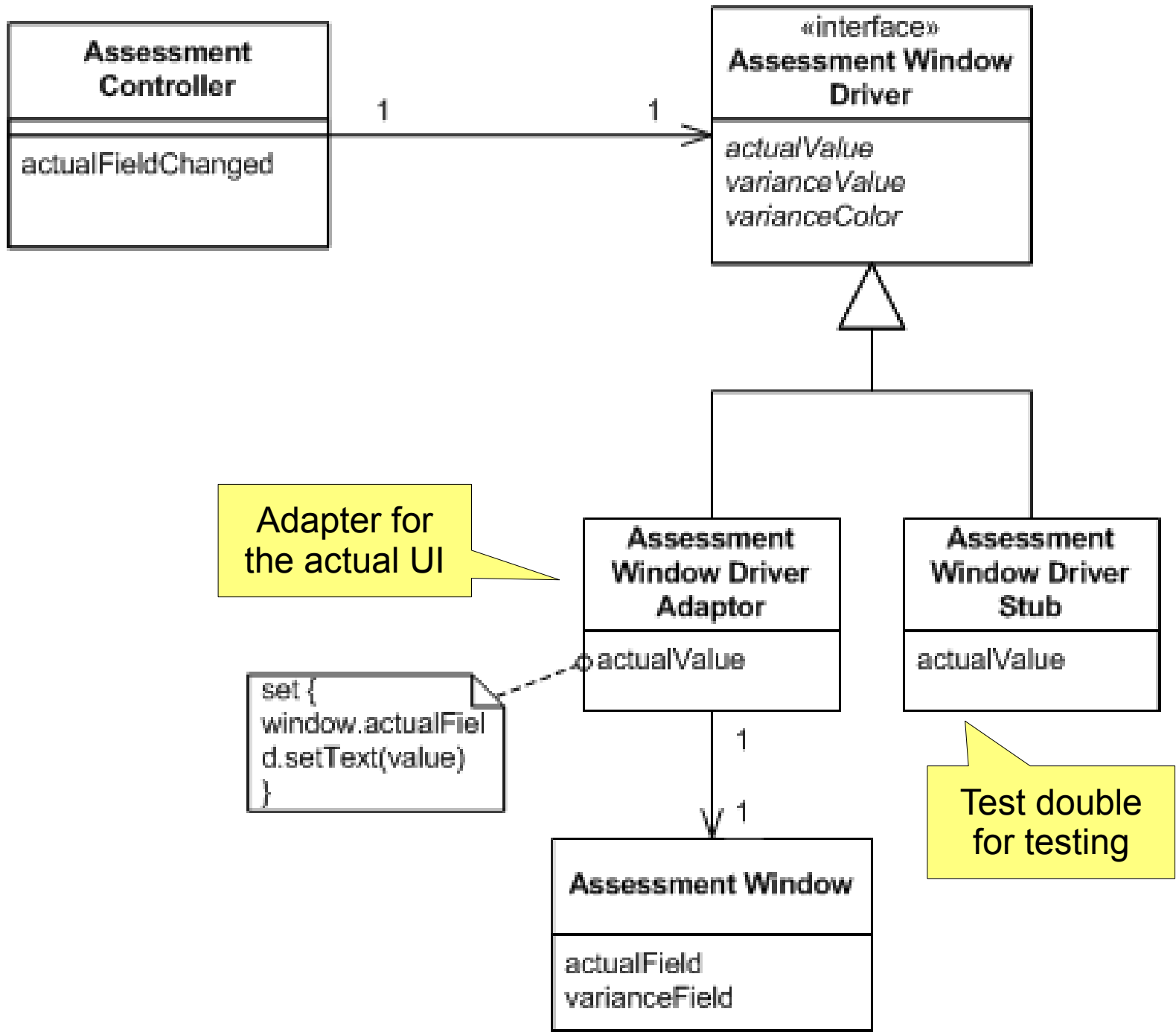
- Use a declarative data binding between the UI view and model for simple view logic (assuming the UI framework supports data bindings).
- Use a controller to handle input response and to manipulate the view to handle more complex view logic.
- Responsibilities of the controller:
 - Input response
 - Partial view/model synchronization



Simple logic done by data bindings

Complex logic done by the controller

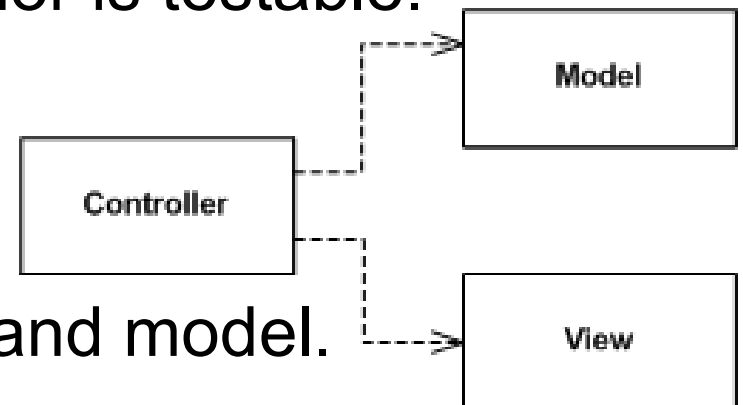


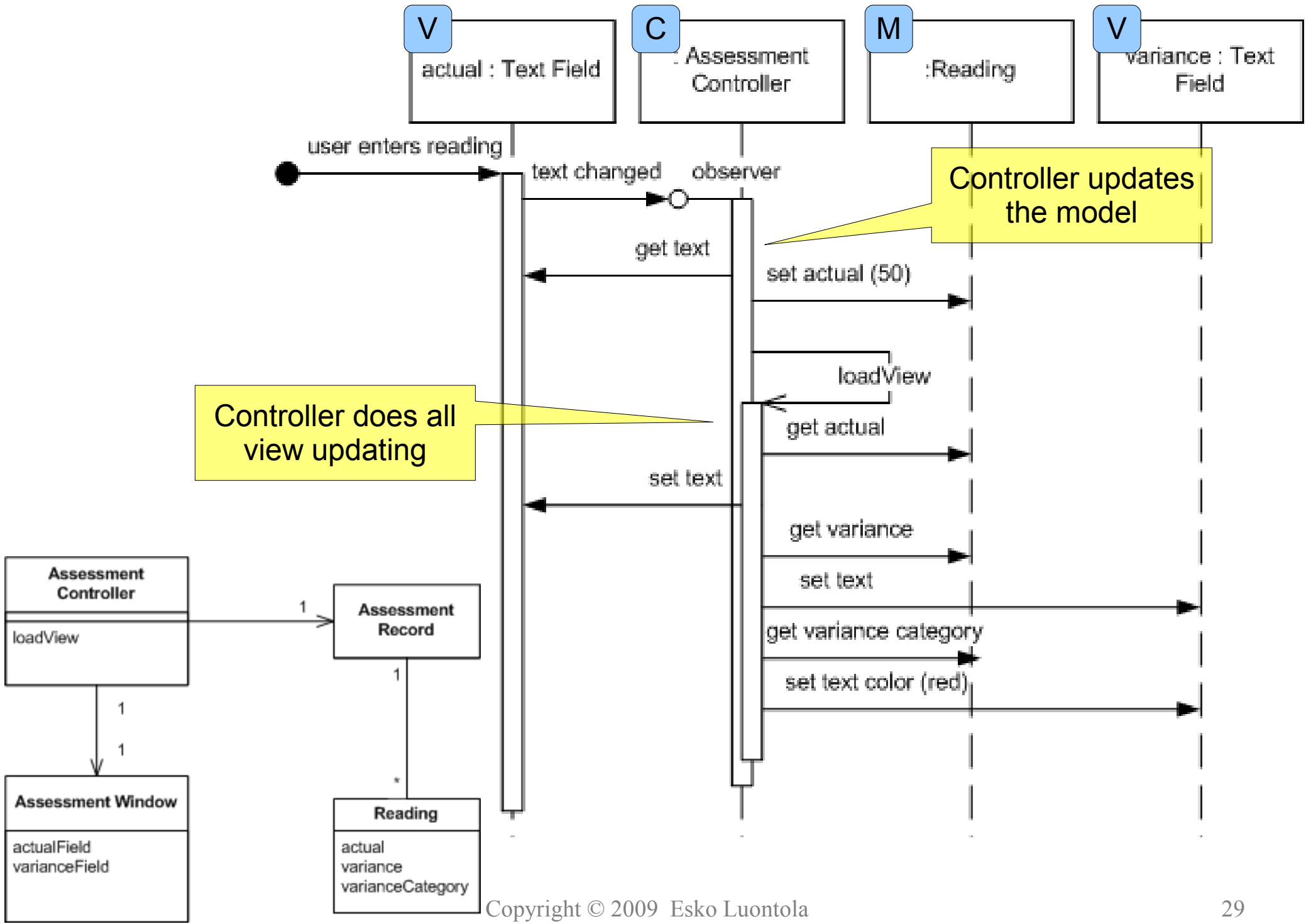


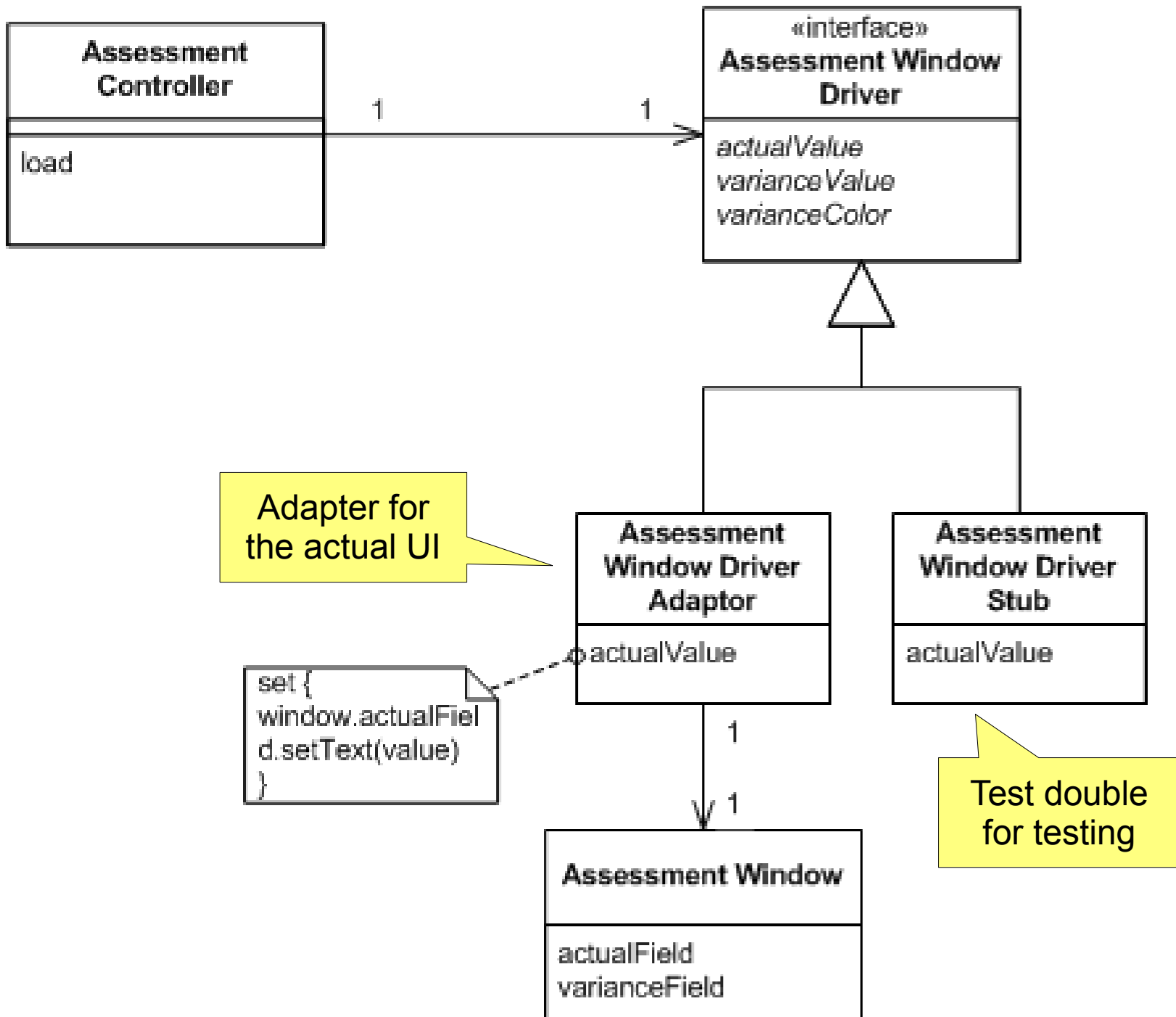
Passive View

A screen and components with all application specific behavior extracted into a controller so that the widgets have their state controlled entirely by the controller.

- Passive View is a very similar pattern to Supervising Controller, but with the difference that *Passive View puts all the view update behavior in the controller, including simple cases*. This results in extra programming, but does mean that *all* the presentation behavior is testable.
- Responsibilities of the controller:
 - Input response
 - Full view/model synchronization
- No dependencies between the view and model.

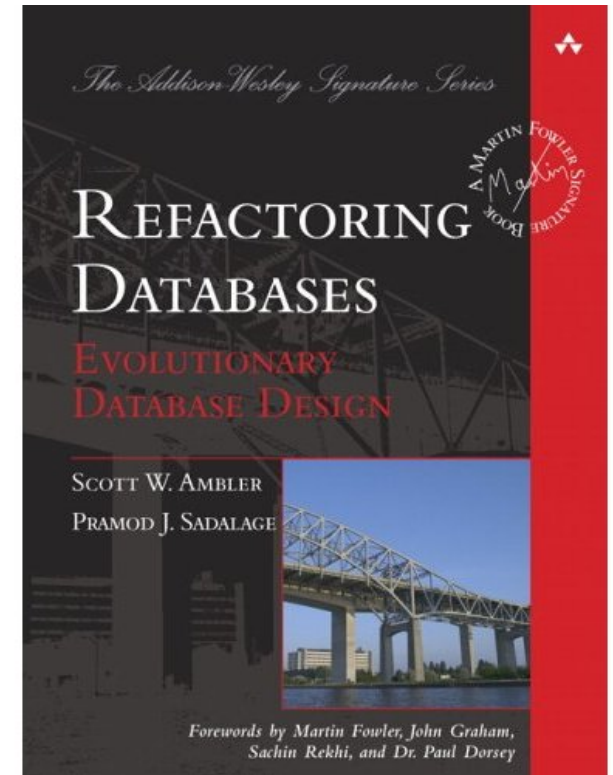






Refactoring Databases

- Allowing the database design to evolve as the application develops is very important for agile methods.
- There are techniques for applying continuous integration and automated refactoring to databases. These techniques work in both pre-production and released systems.



<http://www.amazon.com/Refactoring-Databases-Evolutionary-Database-Design/dp/0321293533>

<http://databaserefactoring.com/>

<http://www.martinfowler.com/articles/evodb.html>

Refactoring Databases: Practices

- DBAs collaborate closely with developers
 - The developer knows what new functionality is needed, and the DBA has a global view of the data in the application.
- Everybody gets their own database instance
 - Developers experiment with how to implement a certain feature and may make a few attempts before settling down to a preferred alternative. It's important for each developer to have their own sandbox where they can experiment.
- Developers frequently integrate into a shared master
 - It's much easier to do frequent small integrations rather than infrequent large integrations. It seems that the pain of integration increases exponentially with the size of the integration.

Refactoring Databases: Practices

- A database consists of schema and test data
 - Tests can assume the test data is in place before they run.
 - Having sample data forces to ensure that database schema changes also migrate the sample data.
- All changes are database refactorings
 - Compared to code refactorings, database refactorings contain three changes that must be done together:
 - Changing the database schema
 - Migrating the data in the database
 - Changing the database access code
 - Individual refactorings are very small.
 - Destructive and complex changes need more care. Big changes might be postponed to the start of next iteration.

Refactoring Databases: Practices

- Automate the refactorings
 - Write the schema changes and data migration as scripts.
 - A database can be updated to the latest version by running all scripts which have been added since the last update.
 - Will be simpler if the database can be taken offline, but upgrading a 24/7 database should also be possible.
- Clearly separate all database access code
 - When there is a database access layer which is clearly separated from the rest of the code, it will be easier for the DBAs to see how the database is used and then optimize accordingly. Also developers will need less knowledge of the SQL queries.

Course Material

- <http://www.objectmentor.com/resources/articles/WorkingEffectivelyWithLegacyCode.pdf>
- <http://martinfowler.com/eaDev/SupervisingPresenter.html>
- <http://martinfowler.com/eaDev/PassiveScreen.html>
- <http://blog.objectmentor.com/articles/2008/06/22/observations-on-test-driving-user-interfaces>
- <http://www.martinfowler.com/articles/evodb.html>