COMP23420- Software Engineering

November 26, 2017

1 Model View Controller Web Frameworks

Mapping MVC into the real world is messy, requires lot of trades offs and there is more than one way to do it.

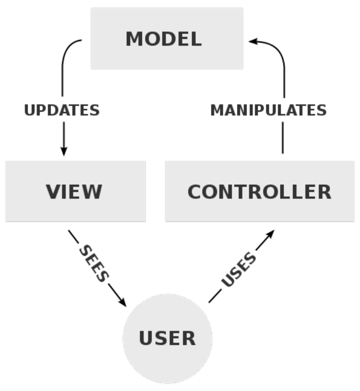

Figure 1: MVC

- **Models** represent knowledge. The nodes of a model should represent an identifiable part of the problem. The nodes of a model should all be on the same problem level.
- A **view** is a visual representation of its model. It acts as a **presentation filter**, highlighting certain attributes of the model and suppress others. A view is attached to its model to get the data and it may also update it. It has to know the semantics of the attributes of the model it represents.
- A **controller** is the link between a user and a system. It provides the user with input and the means for user output. It passes messages onto one or more views.

A controller should never supplement the views. A view should never know about user input.

A controller is connected to all its view. Some views provide an **editor** (which acts as extension of the controller) that permits the user to modify the information that is presented by the view. Once the editing process is completed, the editor is removed from the path and discarded.
MVC was one of the first attempts to do serious UI work on any kind of scale.

The idea behind **separated presentation** is to make a clear division between *domain objects* that model our perception of the real world, and *presentation objects* that are the GUI elements on screen. The domain object is referred to as the model: completely ignorant of the UI. The controller's job is to take the user's input and figure out what to do with it. You have a view-controller pair for each element of the screen, each of the controls and the screen as a whole.

MVC has no event handlers on the assessment controller or the lowest level components.

MVC divide GUI widgets into a controller (for reacting to user stimulus) and view (for displaying the state of the model). Controller and view should not communicate directly but through the model.

- **Observer synchronisation**: have views and controller observe the model to allow multiple widgets to update without needed to communicate directly.

- ⇕ These are different.
- **Flow synchronisation**: each time you do something that changes state that is shared across multiple screens, you tell each screen to update itself.

2 User Interface Design

User stories and use cases are elaborated into scenarios and visual designs ahead of implementation. Details are left out, the rough scope of the application is captured. Flaws are exposed. Same requirements can end up in different designs.

Different fidelity levels: sketches, wire-frames, mock ups, prototypes.

Mock ups:

- Stimulates a dialogue with the costumer to confirm requirements capture, different choices can be shown, exchange of ideas.
- Save time to prevent misunderstandings and remove bugs early on.
- Tips like extracting the tasks from requirements and follow a top–bottom approach.

The **height golden rules of interface design** by Ben Shneiderman:

1. **Strive for consistency**: identical terminology, consistent colour, layout, capitalisation, fonts. Exceptions should be comprehensible and limited in number.
2. **Seek universal usability**: recognise the needs of diverse users and facilitate transformation of content. Adding features for novices and features for experts enriches the interface design and improve perceived quality.
3. **Offer informative feedback**: for every user action, there should be an interface feedback. Visual presentation of the objects of interest provides a convenient environment for showing changes explicitly.
4. **Design dialogues to yield closure**: informative feedback at the completion of a group of actions gives users the satisfaction of accomplishment.
5. **Prevent errors**: if users make an error, the interface should offer simple, constructive and specific instruction for recovery.
6. **Permit easy reversal of actions**: it encourages exploration of unfamiliar options.
7. **Keep users in control**: experienced users strongly desire the sense of being in charge of the interface, it needs to responds to their actions with no surprises.
8. **Reduce short–term memory load**: avoid interfaces in which users must remember information from one display and then remember that information on another display.

3 Data Modelling and Persistence

In Spring, **model** and **entity** are used interchangeably. Spring calls them entities because they are entities in the database. For Spring, a *model* is the data passed to a view.
In MVC think of the model as being made up of a set of entities and **D**ata **A**ccess **O**bjects (**DAO**). In Spring you annotate **P**lain **Old** **J**ava **O**bjects (**POJO**) for persistence.

- `@Entity`: creates a table in the database, a column for each field.
- `@Table`: can be used to give a human–readable name to the table.
- `@Id`: mark the primary key field as a *unique* identifier.
- `@GeneratedValue`: the persistence layer will generate the ID for you when the entity is saved to the database.

- `@Temporal`: to markup dates and times.

`@Entity` and `@Table` are class-level annotations, while `@Id` and `@GeneratedValue` are used for primary key fields.
You can setup relationships between entities:

- `@ManyToOne`: *an* event has *one* venue.
- `@OneToMany`: *a* venue hosts *many* events.
- `@OneToOne`: *a* venue has *one* manager. *A* manager manages only *one* venue.

Repositories are the Spring mechanisms for querying the underlying database, data lives in repository.

CRUD means **C**reate **R**ead **U**pdate **D**elete and are repository implementations.
A **marker interface**, Spring magic creates a concrete implementation.
It uses a `Service` to call out a repository.
CRUD provides: `count()`, `findOne(long id)`, `findAll()`, `save(Venue venue)`, `delete(long id)`.
Create a `Service` interface to expose the methods you need and get Spring to auto-wire the repository. Spring has a mechanism to convert method names into queries: simply define the queries as methods in the repository interface, like `public Venue findFirstByNameOrderByNameAsc(String name)`.

4 MVC Architectural Design Pattern

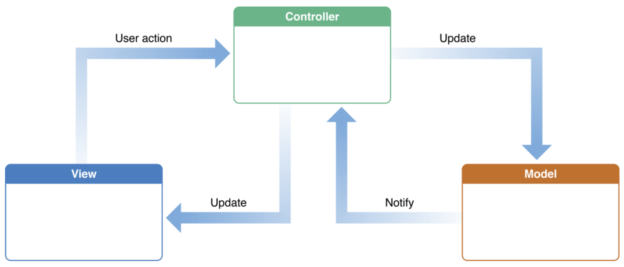

Figure 2: MVC

The MVC design pattern assigns the objects in an application with the role of model, view or controller. The collection of objects that play a certain MVC role is sometimes referred to as a **layer**, as in in a **layered architecture**.
The MVC pattern also specifies how objects should communicate with one another. Different ways of communication result in different variants of the MVC pattern.

Model objects encapsulate the data specific to an application and define the logic and computation that manipulates and processes that data. They can be reused in similar applications. It can have `@OneToOne` or `@OneToMany` relationships with other model objects.

Model objects should not directly communicate with the View objects, as they should not be concerned with user-interface and presentation issues. When a model object changes, it notifies a Controller object, which updates the appropriate View objects.

View objects present the data from the application's model objects and enable the users to view or edit data. They are typically reused to provide consistent look-and-feel interfaces. They communicate with the controller objects to learn about changes in the data and to notify user–initiated changes.

Controller objects act as an intermediary between the View and Model objects, thus they communicate with both types. They serve as a **conduit**. They can also perform setup and coordinating takes for an application and manage the life cycle of other objects.

MVC is central to a good design for a Cocoa application. Some benefits:objects are more reusable, interfaces are better defined, applications are more easily extensible. Cocoa is Apple's native object–oriented API for macOS.

Document architecture uses three classes: `NSDocument`, `NSDocumentController`, `NSWindowController`. Objects of these classes are arranged into a one-to-many relationship: an `NSDocumentController` object (a *singleton*, only one instance at any one time) creates and manages one or more `NSdocument` objects; a `NSDocument` object in turn, creates and arranges one or more `NSWindowController` objects. The **model–controller/coordinating controller** object creates, opens, tracks and manages documents and manages the Open Recent Menu. A **view–controller** object manages one windows associated with a document. if a document has multiple windows, each window has its own view–controller which is responsible for closing windows after ensuring that data is saved. View objects collectively produce a hierarchy of windows. A `NSDocument` object is a container of data and:

1. Owns and manages data in memory when a document is opened and writes the data to a disk file under a user assigned name.
2. It displays the data in one or more windows and makes it available for editing.
3. Can print, revert and undo operations on the data.

Persistent data objects are stored in a database. Model objects and their relationships are represented using E-R Diagrams.
The Cocoa Core Data framework supports object modelling. You can also represent the values of objects.
MVC is crucial to designing Chrome Apps, along with Angular JS, Kendo UI and Sencha.
Benefits:

- Programs are more reusable.
- Interfaces tend to be better defined.
- More adaptable to changing requirements, so more easily extensible.
- MVC requires that objects in a program play one of the roles defined by MVC, making it clear which object should perform which tasks.
- Promoting good design principles like separation of concerns, higher cohesion and lower coupling and design for change.

Other architectural design patterns, such as Layered Architecture, Presentation–Abstraction–Control, and Publisher–Subscriber, Model–View–Presenter, are all variants of MVC.

Minimum Viable Product is the basic implementation that contains the minimum number of features required to provide a demonstration of the product's worth. It can be continually built on up to the release, so that you always have a working version that you can show to a customer, enabling you to get valuable feedback on whether development is progressing as expected.

Spring can parse dates and times (using parse helper) from forms with the help of another annotation on the field in your entities.
Boostrap includes some nice form controls to help the user input dates and times in sensible formats.

5 Specification by Example

Test data needs to be comprehensive, correct, and cover all the requirements.
Specification by example can help to ensure that software unambiguously meets requirements.

6 Testing Functionality in Isolation

Unit tests:

- Tests do not build on other tests.
- Test one *only* thing.
- Stay within class/process/network boundaries, do not test the database layer as a side effect.

In order to test things together we use **integration tests**, where we rest the system from end to end, and **system tests**.

In order to perform unit testing on a controller, it must be isolated from everything else like DAO, database, views. Spring includes a mocking library in its dependencies (**Mockito**, invoked using `@Mock`). DAOs are `@Autowired` within a controller. We need to inject mocks into the controller to mock out the Data Access Objects.
Using `@Mock`, marks an object for mocking: now all methods that return a value return null.
To solve this problem we use `when(<condition>).then<do something>`. We can check if a method is called a fixed number of times with `verify(VenueService, times(1)).methodToTest(args)` or `verify(VenueService, atMost(5)).methodToTest(args)` or `verify(VenueService, never()).methodToTest(args)`.

Test coverage is a useful tool for finding untested parts of a codebase.
You do enough testing if:

- You rarely get bugs that escape into production.
- You are rarely hesitant to change some code for fear it will cause production bugs.

You are testing too much if you can remove tests while still having enough of them.

There are different ways to test something:

- **Dummy**: passed around but never actually used.
- **Fake**: has working implementation but usually take some shortcut.
- **Stub**: provide canned answers to calls made during the test.
- **Spies**: stubs that record information based on how they were called.
- **Mock**: pre-programmed with expectations which form a specification of the calls they are expected to receive.

7 Using External APIs: Google Maps API

Looking for the right API is a search problem.
Looking for the right functionalities is a search problem.

Geocoding: mechanisms by which you get longitude and latitude from the address.
Address lookup: mechanism to get addresses from geographical coordinates.
The **JavaScript API** plots markers on locations if longitude and latitude are given.
The **Web Service API** is for using geocoding functionalities in the server side, while the **Web API** on the client side display maps and markers.

Programming by example: practice of reusing existing code on your code. It is an about finding analogies and see the code as a template. There are two antagonistic strategies: copy and paste and try to make it work until it works or try to understand the whys and hows.

8 Integrating External Services: Spring Social Media

Spring Social is a framework of Spring boot. It establishes connections between Spring boot apps and **SaaS** (Software as a Service so API, resources and interface) providers including Facebook, Twitter, LinkedIn, Githun and Tripit.
Role of social integration 101, three way conversation between service provider, service

consumer and the user.
With Spring Social the consumer uses the service provider on behalf of the user.
Authentication: connection and sign in.

9 Acceptance Testing

An **acceptance test** (also known as **functional test** or **consumer test**) is a formal description of the behaviour of a software product. It generally has a binary result, with either pass or fail. A failure **suggests** the presence of a defect in the product, but it *does not prove* it.
Benefits:

- Encouraging closer collaboration between developers and customers, users, domain experts.
- Providing a clear and unambiguous contract between customers and developers.
- Decreasing the chance and severity both of new defects and regressions.

Acceptance tests that are unduly focused on technical implementation also run the risk of failing due to minor changes which in reality do not have any impact on the product's behaviour.
Costs:

- The creation of acceptance tests requires significant effort.
- The maintenance of tests is burdensome.
- Sometimes customers cannot understand them.

10 RESTful Web Services and Test–Driven Development

REpresentational **S**tate **T**ransfer, or **REST** is an architectural style. It is the underlying architectural principle of the World Wide Web.
Clients like browsers can operate without knowing anything about the server and the resources it hosts.

Server and client must both agree on the media used, like HTML for the Web. Resources are identified by a persistent identifier, like URIs. Resources are manipulated using a common set of verbs, like CRUD becomes POST, GET, UPDATE, DELETE.

The representation retrieved for a resource is dependent on the request and not the identifier. Use Accept headers to control whether you want HTML, XML, JSON, ...

It embeds the relationships between resources in the representation of the resource, like links between objects are embedded directly in the representation. **Resource representations** describe how the representation can be used and under what circumstances it should be discarded/re-fetched in a consistent manner.

Some HTTP verbs are **idempotent** like GET, PUT, DELETE. This means that they are free from side-effects.
REST applications provide responses that are self descriptive and link to other resources as necessary.

10.1 Test Driven Development

With test driven development the development is driven by the tests in order to **provide feedback**.

1. Write a simple test that does not work.

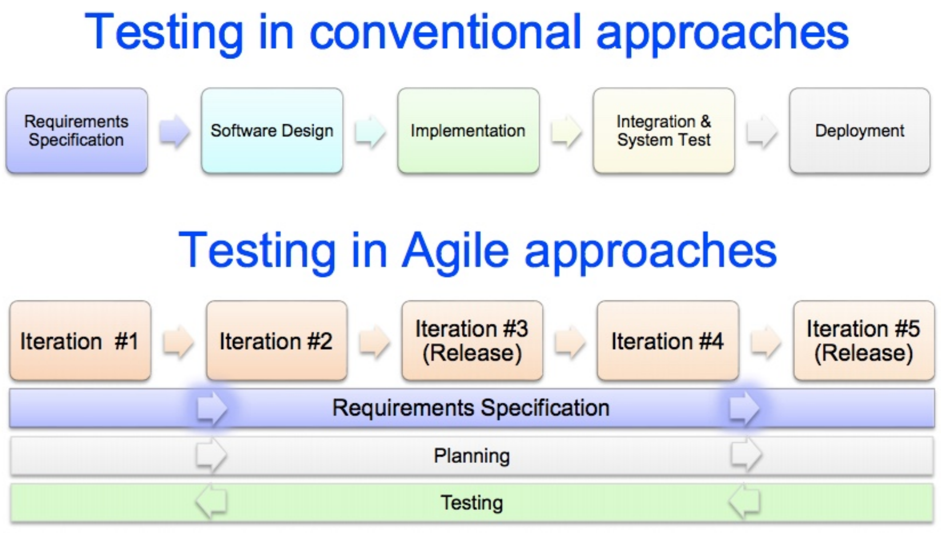
Testing in conventional approaches
Requirements Specification
Software Design
Implementation
Integration & System Test
Deployment
Testing in Agile approaches
Iteration #1
Iteration #2
Iteration #3 (Release)
Iteration #4
Iteration #5 (Release)
Requirements Specification
Planning
Testing

Figure 3: Test Driven Development process

2. Implement only that which is required to pass the test, make the test work, quick and dirty.
3. Refactor: improve code quality and eliminate any duplication.
4. **Repeat**.

Rules:

- Write new code only if a test has failed.
- Eliminate duplication.

Implications:

- Design must be organic, running code provides feedback between decisions.
- Developers must write the unit tests, they cannot wait for someone else to do so.
- Developer environment must provide rapid feedback after small changes.
- System must consist of many loosely coupled components to make testing easy.

Motivation:

- **Q**uality **A**ssurance becomes proactive.
- Estimations can be accurate enough to involve real customers in daily development.
- Engineers can engage in minute-by-minute collaboration.
- Short iterations deliver real business value.
- Way of managing fear during programming.
- Encourages good OO design practice.
- Encourages design for testability.
- We get an unambiguous progress meter.
- We build up a comprehensive set of regression tests as we go along.

Limitations:

- Conceptually steep learning curve, especially at the start of the project.
- Can all programming tasks be driven by tests?