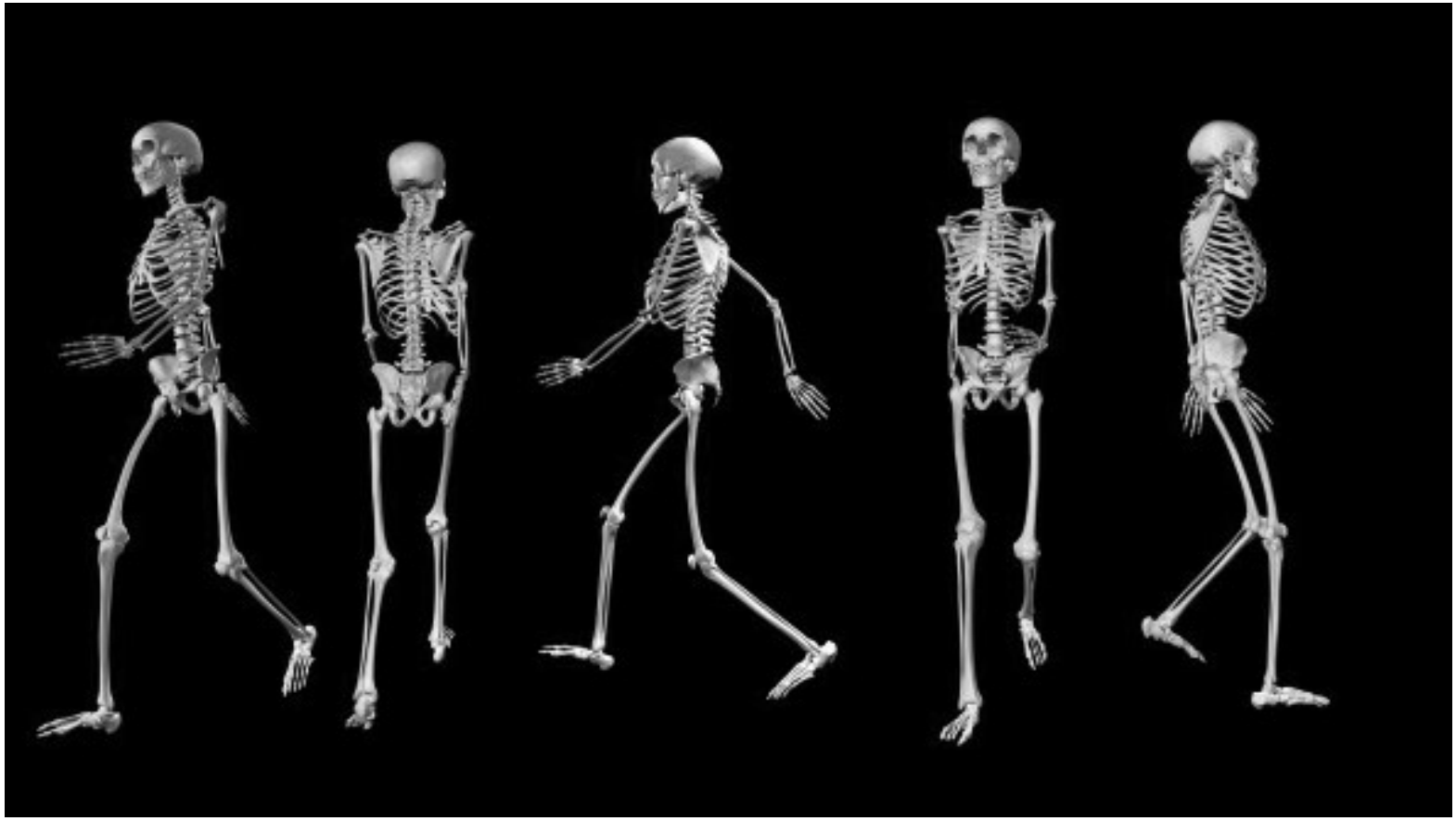
Assignment 2 Bootstrap

The API + Command Line Specification

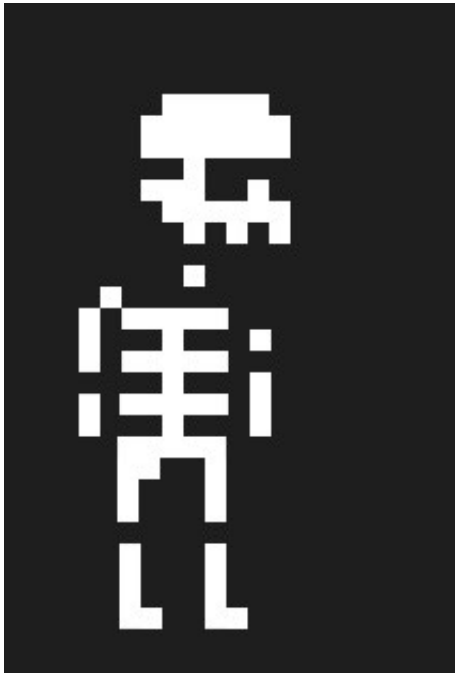
```
addUser(firstName,lastName,age,gender,occupation)
removeUser(userID)
addMovie(title, year, url)
addRating(userID, movieID, rating)
getMovie(movieID)
getUserRatings(userID)
getUserRecommendations(userID)
getTopTenMovies()
load()
write()
```

Walking Skeleton



Walking Skeleton : http://alistair.cockburn.us/Walking+skeleton

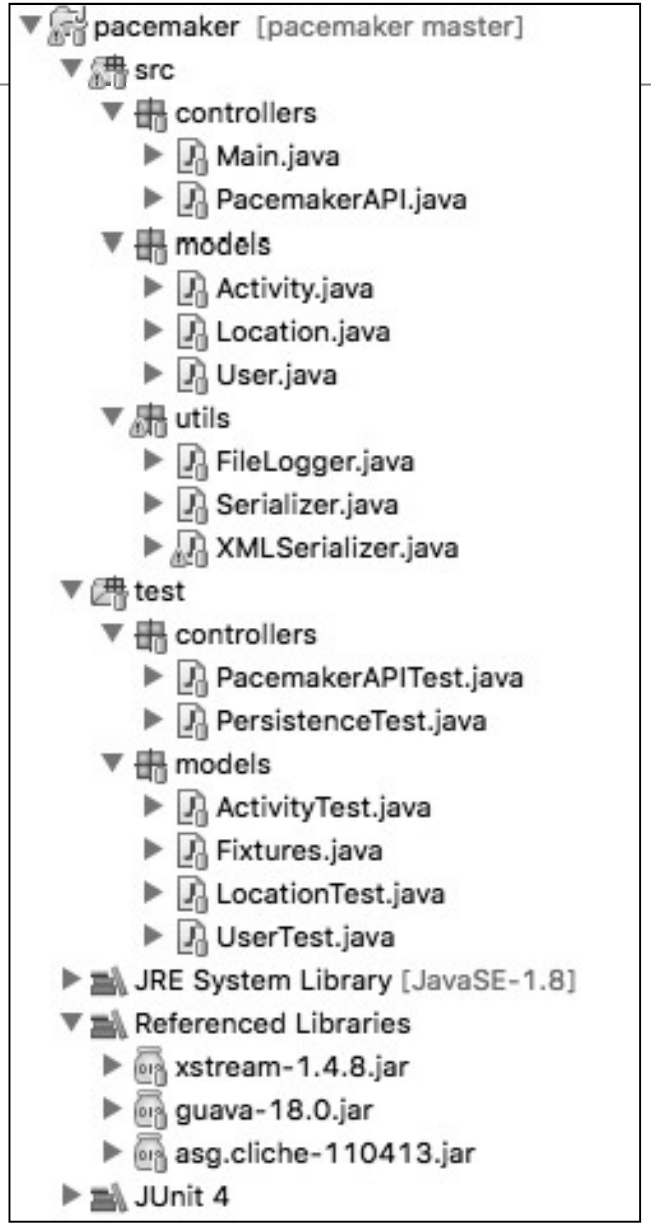
- “A Walking Skeleton is a tiny implementation of the system that performs a small end-to-end function.
- It need not use the final architecture, but it should link together the main architectural components.
- The architecture and the functionality can then evolve in parallel.”



- Useful strategy to get started…
- Enables you to make small incremental improvements from a solid foundation

Walking Skeleton Project

- Create an Eclipse project with:
  - "src" and "test" folders
    - best guess at suitable packages
    - Initial versions classes you think you will need - largely empty for the first version
  - Libraries



Building a Walking Skeleton for Assignment 2

- What are the likely initial Objects?
- What data structures will be appropriate?
- How will the API be implemented?
- What strategy is to be used for the command line?
- How will the tests be organised?
- What is the persistence strategy?

```
addUser(firstName,lastName,age,gender,occupation
removeUser(userID)
addMovie(title, year, url)
addRating(userID, movieID, rating)
getMovie(movieID)
getUserRatings(userID)
getUserRecommendations(userID)
getTopTenMovies()
load()
write()
```

Give the Project a Name!

CODING HORROR
programming and human factors

12 Nov 2007

What's in a Project Name?

Since I started at Vertigo, here are a few of the projects I've worked on:

- Michelangelo
- Nash
- Whiskeytown
- Gobstopper

These are our **internal project code names**. The names are chosen alphabetically from a set of items; every new project gets a name from the set. We start with A, and when we finally arrive at Z, we pick a new set of items for project name inspiration. Can you guess which set each of the above project names is from? No cheating!

We've come up with the following loose guidelines for project naming:

1. We prefer one word names.
2. They should be relatively easy to pronounce and easy to spell.
3. They have to be client friendly.
4. They should be globally unique across the company. No duplicates.
5. We need a reasonable number of items in the set to choose from, in A-Z order.

Types of Food
Video games (Atari 2600, Arcade, etc)
Brands of Beer
Roman Emperors
Cartoon characters / shows
Mythological names / gods
Cars
GUIDs (a personal favorite)
Gemstones
Types of Coffee drinks
States
Counties
Plants
Hitchcock films
Dog breeds
Colors
Famous Explorers
Trees
IRS Tax Forms
English monarchs
Famous People (eg, Sagan)
Wikipedia article names
Single letters (including unicode)
Radio alphabet
Candy brands
Dinosaurs
Historical Sites
City street names
IKEA product names
Types of Fasteners (nut, bolt, rivet, etc)
Ski resorts
National Parks
Mountain Peaks
World War II era ships
Birds
Beaches
Bridges
Web 2.0 names
Warcraft realm names
Cheeses
Countries
Cereal brands

likemovie

- What are the likely initial Objects?
- What data structures will be appropriate?
- How will the API be implemented?
- What strategy is to be used for the command line?
- How will the tests be organised?
- What is the persistence strategy?
- Where do I start?

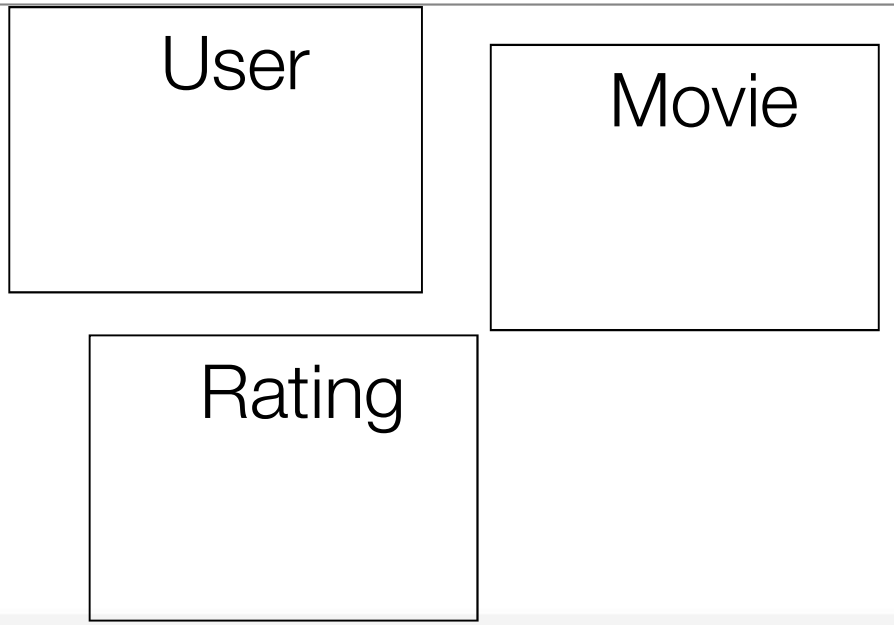
Initial Candidate Objects

- Model
  - User
  - Movie
  - Rating
- LikeMoviesAPI
- CommandShell
- Serialisers?

```
addUser(firstName,lastName,age,gender,occupation)
removeUser(userID)
addMovie(title, year, url)
addRating(userID, movieID, rating)
getMovie(movieID)
getUserRatings(userID)
getUserRecommendations(userID)
getTopTenMovies()
load()
write()
```

What data structures will be appropriate?

- Map of userId->User
- Map of movieId->Movie
- Ratings?
  - Each user holds a list of ratings objects




```
addUser(firstName,lastName,age,gender,occupation)
removeUser(userID)
addMovie(title, year, url)
addRating(userID, movieID, rating)
getMovie(movieID)
getUserRatings(userID)
getUserRecommendations(userID)
getTopTenMovies()
load()
write()
```

How will the API be implemented?

- Define a single LikeMoviesAPI class
- Define the data structured (userIndex, moviesIndex) as members of the API class
- Define a suitable method signature for each of the features listed here
- API does not include any UX

```
addUser(firstName,lastName,age,gender,occupat
removeUser(userID)
addMovie(title, year, url)
addRating(userID, movieID, rating)
getMovie(movieID)
getUserRatings(userID)
getUserRecommendations(userID)
getTopTenMovies()
load()
write()
```

Command Line

- DON’T roll your own
- Use a suitable library

Cliche Command-Line Shell

Cliche is a small Java library enabling *really* simple creation of interactive command-line user interfaces.

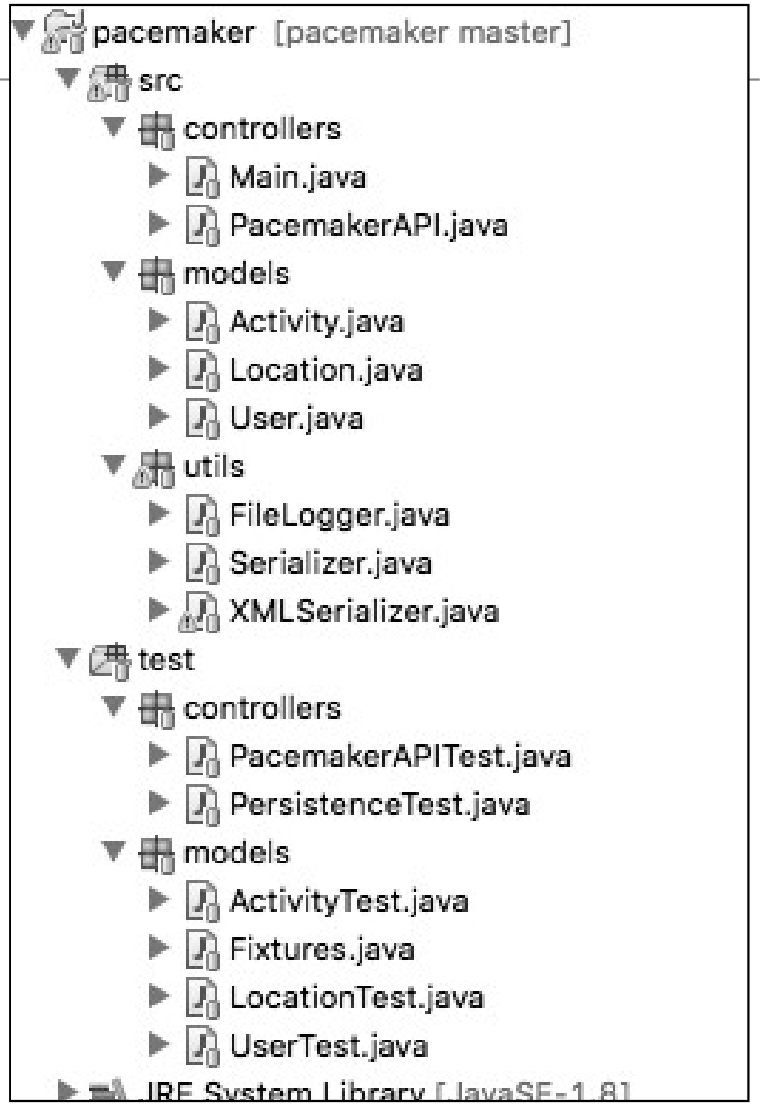
It uses metadata and Java Reflection to determine which class methods should be exposed to end user and to provide info for user. Therefore all information related to specific command is kept in only one place: in annotations in method's header. User don't have to organize command loop, write complicated parsers/converters for primitive types, though he can implement custom converters when needed.

How simple? *So simple:*

```
package asg.cliche.sample;

import asg.cliche.Command;
import asg.cliche.ShellFactory;
import java.io.IOException;

public class HelloWorld {

    @Command // One,
    public String hello() {
        return "Hello, World!";
    }

    @Command // two,
    public int add(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) throws IOException {
        ShellFactory.createConsoleShell("hello", "", new HelloWorld())
            .commandLoop(); // and three.
    }
}
```

Testing

- Create a separate top level ‘source folder’ for all tests
- Mirror the ‘src’ package structure in this folder
- Create one test class for each ‘src’ class



Persistence

- Use a suitable high level library
- Single file to store entire object model
- Alternatives?

About XStream

XStream is a simple library to serialize objects to XML and back again.

Features #features

- **Ease of use.** A high level facade is supplied that simplifies common use cases.
- **No mappings required.** Most objects can be serialized without need for specifying mappings.
- **Performance.** Speed and low memory footprint are a crucial part of the design, making it suitable for large object graphs or systems with high message throughput.
- **Clean XML.** No information is duplicated that can be obtained via reflection. This results in XML that is easier to read for humans and more compact than native Java serialization.
- **Requires no modifications to objects.** Serializes internal fields, including private and final. Supports non-public and inner classes. Classes are not required to have default constructor.
- **Full object graph support.** Duplicate references encountered in the object-model will be maintained. Supports circular references.
- **Integrates with other XML APIs.** By implementing an interface, XStream can serialize directly to/from any tree structure (not just XML).
- **Customizable conversion strategies.** Strategies can be registered allowing customization of how particular types are represented as XML.
- **Security framework.** Fine-control about the unmarshalled types to prevent security issues with manipulated input.
- **Error messages.** When an exception occurs due to malformed XML, detailed diagnostics are provided to help isolate and fix the problem.
- **Alternative output format.** The modular design allows other output formats. XStream ships currently with JSON support and morphing.

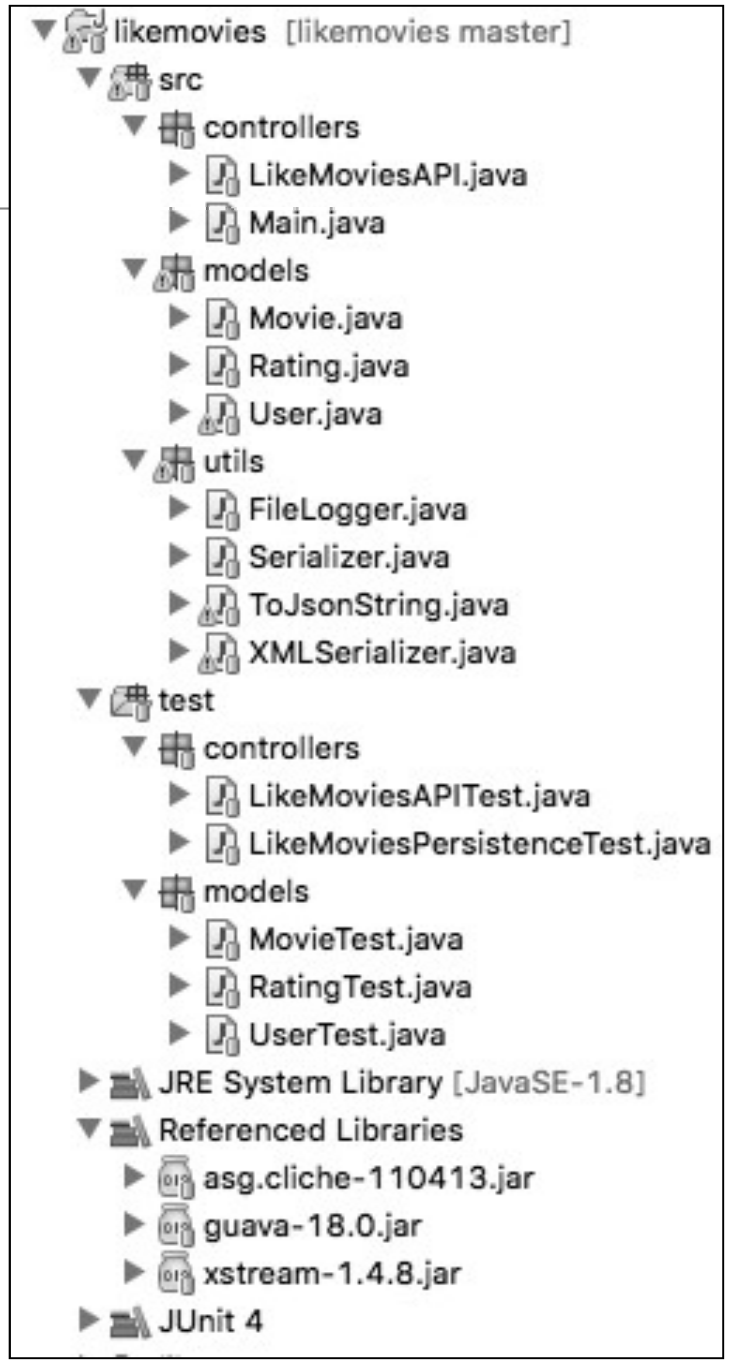

where do I start?

Plan for Assignment 2

- Build Walking Skeleton
- Break features into simple ‘Stories’
- Reorder the stories into simplest to implement first
- For each story:
  - Write a test
  - Implement the necessary features
  - Implement and verify the command

Build Walking Skeleton

- Define very simple model objects : User & Movie
- Implement tests for these
- Implement Simple command for add and list all users
- Create skeleton version classes you think you will need (even if they are empty)

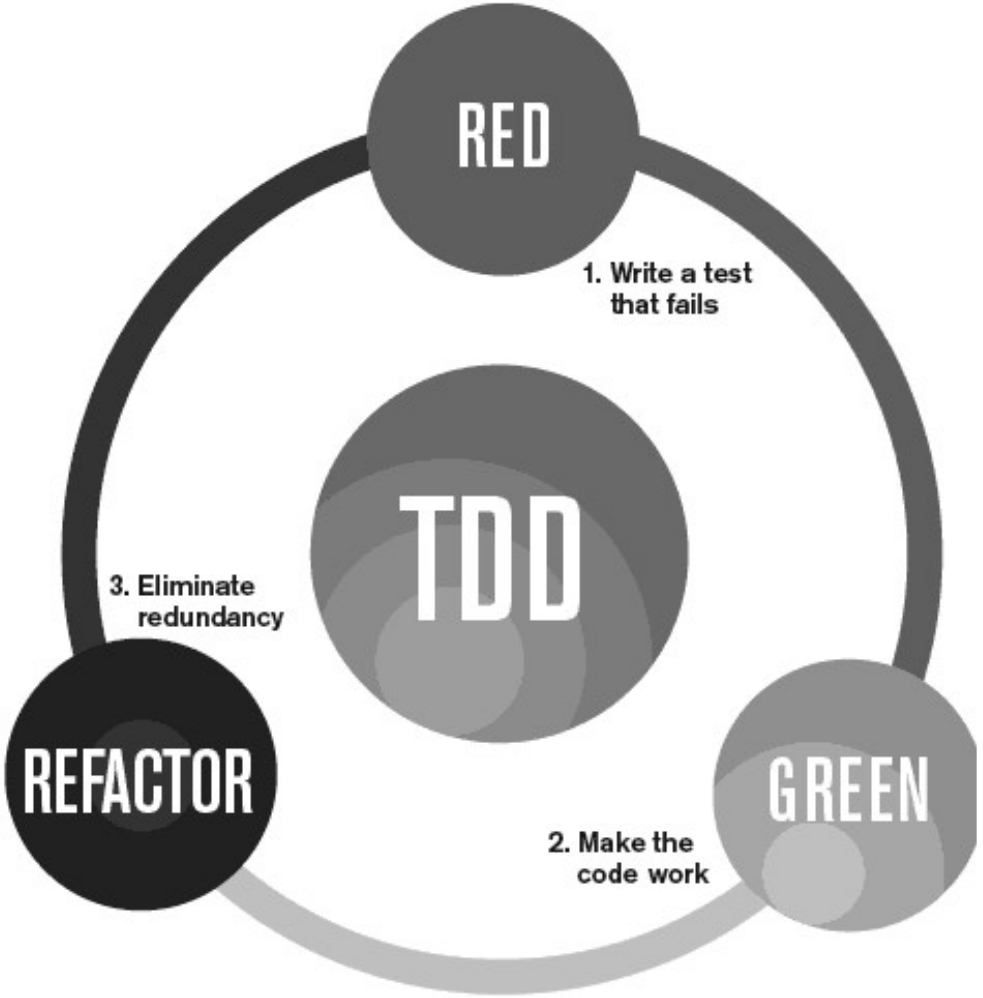


Stories

- add a user
- add a movie
- remove a user
- get a movies details
- rate a movie
- get a users ratings
- get the top ten movies (by all ratings)
- for a given user, get recommendations for that user (recommendation algorithm)
- read the movie db from an external css file
- save / load the main application model

Organise Stories

- Projects are usually implemented in ‘Iterations’
- Each Iteration starts with a subset of stories the iteration will tackle
- For each story in the iteration:
  - Write a test
  - Implement sufficient features for the test to pass
  - Refactor to improve the implementation



The mantra of Test-Driven Development (TDD) is “red, green, refactor.”

Iteration Plan (suggestion)

- Iteration I
  - add a user
  - add a movie
  - remove a user
  - get a movies details
- Iteration II
  - rate a movie
  - get a users ratings
- Iteration III
  - save / load the main application model (to XML or JSON)
  - read the initial movie db from an external csv file
- Iternation IV
  - get the top ten movies (by all ratings)
  - for a given user, get recommendations for that user (recommendation algorithm)

Walking Skeleton - Extracts

- This skeleton closely modelled on pacemaker
- the ‘utils’ package can be carried over as is
- API and Main mirror the pacemaker organisation:
  - API - no UI, just manage the data
  - Main - deal with all UI via cliche