Emergent Design

TDD as a design tool

Design?

- Classes
- Object interaction
- Structure of soft layers (UI, MVC, API, Services, Data access, delegates etc.)
- Data structures
- Algorithms
- Patterns

Answering how does this feature works?

vs. Architecture

- Infrastructure
- Application layers
- Integration positioning
- All things difficult to change later, but
- Need to be decided up-front
- DevOps efforts are making this aspect more flexible

Answering how this software works?

Design First

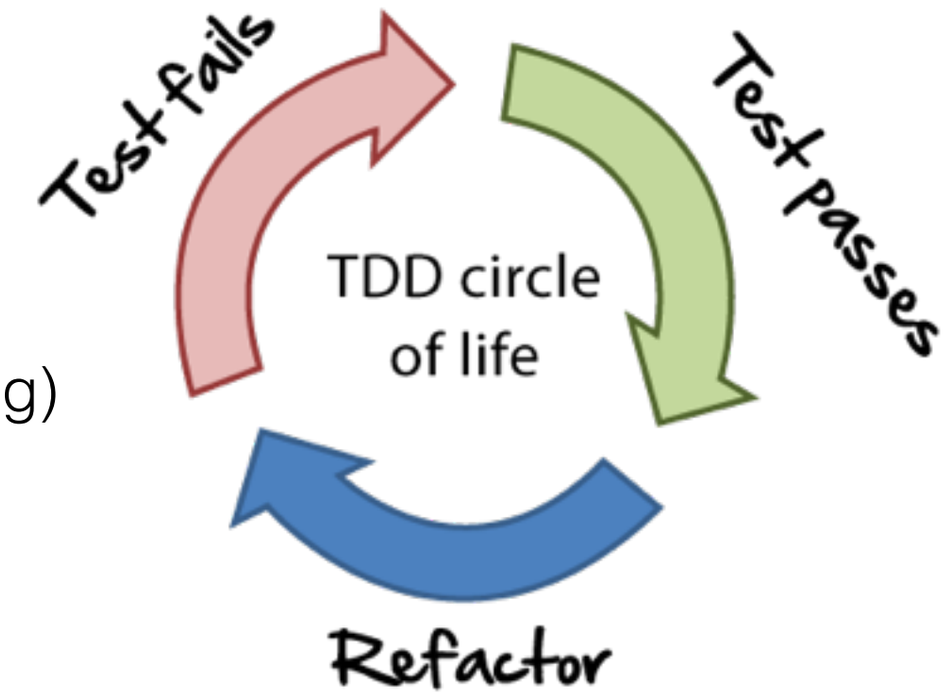
- Understand the requirement of the feature
- Think about what code will be required to build the feature
- Discuss and agree with peers/leads and document it
- Write code which conforms to this design
- Write unit and integration tests
- Ensure all tests pass and deliver the feature

BUFD or BDUF

Emergent Design

- Understand the requirements of the feature
- Find the simplest acceptance criteria for this feature
- Create a test for this acceptance criteria
- Write the test and let it fail (not compiling is failing)
- Write minimum code to pass the test
- Check for refactoring opportunities
- Repeat for the next acceptance criteria

Design emerges as system is built



Why Emergent Design?

- Avoids over engineering (YAGNI)

- Adapts naturally to changing requirements
  - Avoids upfront commitment to one specific pattern

- Collective ownership of design
  - Code does not have a single owner
  - Team members are fungible

- Keeps design as simple as possible

- Results in highly maintainable, less smelly code primarily because of key enablers (TDD, pairing etc.)

Pitfalls

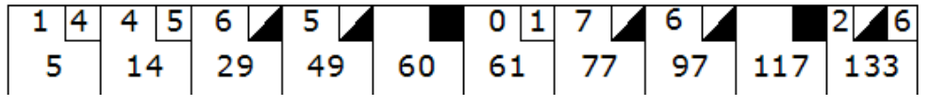
- Team members should be highly skilled and at roughly the same level of competence
  - Vastly different styles of coding
  - Good understanding and experience of code and design smells
- Complex systems with legacy integration needs upfront design
- Distributed teams need common understanding of design
- Wrong implementation will still result in brittle systems
- Can be slow to start with as is counter intuitive to traditional way of coding

Generally ED works very well with teams following XP practices strictly

Adaptive Design

- Harnesses power of ED with light design up-front
- Design elements are “malleable”
- Elements are appropriately decoupled
- Empowers developers to decide lower level implementation
- Coupled with XP practices like pairing, TDD, refactoring to work efficiently

Bowling Game



- The game consists of 10 frames as shown above. In each frame the player has two opportunities to knock down 10 pins.

- The score for the frame is the total number of pins knocked down, plus bonuses for strikes and spares.

- A spare is when the player knocks down all 10 pins in two tries. The bonus for that frame is the number of pins knocked down by the next roll.

- So in frame 3 above, the score is 10 (the total number knocked down) plus a bonus of 5 (the number of pins knocked down on the next roll.)

- A strike is when the player knocks down all 10 pins on his first try. The bonus for that frame is the value of the next two balls rolled.

- In the tenth frame a player who rolls a spare or strike is allowed to roll the extra balls to complete the frame. However no more than three balls can be rolled in tenth frame.

Eclipse setup

- Eclipse Luna for JEE developers
- Infinitest: Constantly checks the workspace for any new tests/ changes to the codebase which may impact the built tests and gives immediate feedback as a coloured bar
- Findbugs and PMD static code analysers
- JaCoCo for visible code coverage within eclipse
- Following java editor favourites have been defined
  org.hamcrest.Matchers.*
  org.hamcrest.CoreMatchers.*
  org.junit.*
  org.junit.Assert.*
  org.junit.Assume.*
  org.junit.matchers.JUnitMatchers.*

Project setup

- Simple Java project
- Dependencies managed by maven
- Following dependencies have been defined junit 4.12, mockito-all 1.10.19, hamcrest-all 1.3