What is Incremental Design?

In brief: Allowing design to arise through fast-feedback practices such as Test-Driven Development, and Refactoring, and a heightened sensitivity to "code smells." Relies on the notion of emergence…

Emergence

> *…a process whereby larger entities, patterns, and regularities arise through interactions among smaller or simpler entities* ***that themselves do not exhibit such properties.***
>
> -- http://en.wikipedia.org/wiki/Emergence

Properties of emergent behaviors in systems:

- From simple rules, complex behaviors manifest.
- Order can arise spontaneously from seemingly chaotic systems.
- "Self-similarity" at all scales: Similar appearance, rules, and/or pathways apply at macro & micro levels.
- Tiny actions, repeated many times, perhaps recursively.
- Requires a rich supply of resources: time, food, energy, RAM, or creativity…

Kent Beck's Four Rules of Simple Design

1. Passes all the _____.

2. Expresses every _____ required.

3. Says everything _____ and _____ _____.

4. Has no _____ parts.

Refactoring

"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written."

> -- *Refactoring: Improving the Design of Existing Code,* Martin Fowler, Addison-Wesley, 1999

Rob's terse definition:

Any change to the _____

that preserves _____ behavior

and improves _____.

A Taste of Code Smells

```
public PrintFormattedAddress(Address a)
{
    PrintLn(a.Line1());
    PrintLn(a.Line2());
    Print(a.City());
    Print(", ");
    Print(a.State());
    Print(" ");
    PrintLn(a.PostalCode());
}
```

The smell: An object is calling another over and over, rather than delegating.

What would be some typical refactorings to fix this?

Follow your nose…

```
class Address
{
    ...
    public string Line1() { return line1; }
    public string Line2() { return line2; }
    public string City() { return city; }
    public string State() { return state; }
    public string PostalCode() { return zip; }
}
```

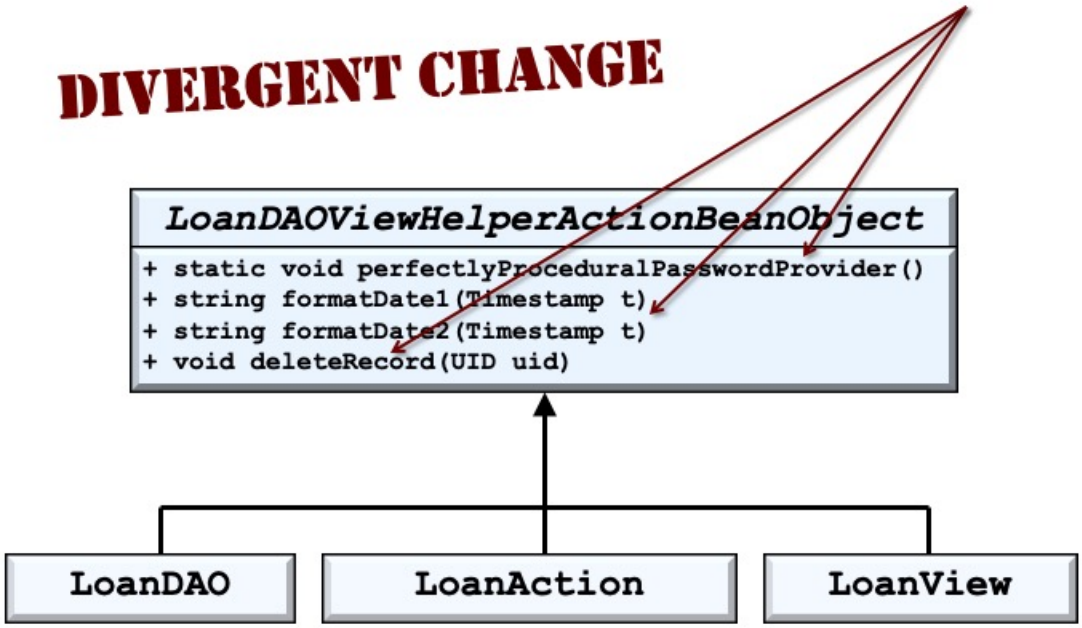
DATA CLASS

The smell: A class isn't doing much else besides giving away its data.

What would be some typical refactorings to fix this?

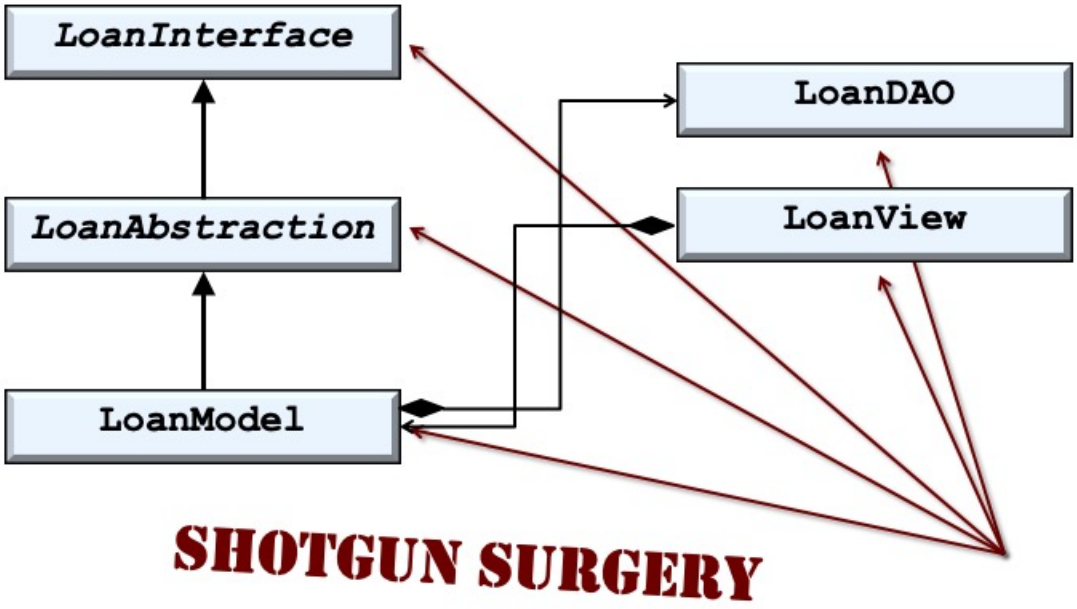
Is this smell possibly related to the smell on the previous page?

Overarching putrescence…



The smell: One class changes in different ways for different reasons. There's often a cohesion/**S**RP problem. (Example above is loosely based on a real system, from a now-defunct business.)

What would be some typical refactorings to fix this?



The smell: A single change to the system requires changes in a number of classes (or files).

What would be some typical refactorings to fix this?

Setup for Refactoring Lab

1. Browse to this github account and find the appropriate `labs-<language>-<unit test framework>` repository:

   `https://github.com/AgileInstitute?tab=repositories`

2. Within your lab folder from the github repo, locate the ***TestedTrek*** subfolder. *This is the only folder you will be using in this lab.*
3. Run all the tests in the folder.

Part a: Refactor TestedTrek for understanding

1. ***Run all the tests.*** They should all _____ .
2. Freely refactor the “fire weapon” or “process command” method on the Game class. Refactor in any way that makes intuitive sense to your pair/trio/mob. Take a look at *https://refactoring.com/catalog/* for inspiration.
    a. You must ***not***, however, alter *untouchables*.
    b. While refactoring, run the tests frequently. You shouldn’t have to change the TestedTrek tests at all, and they ***must continue to pass.***
    c. Perhaps try *Extract Method, Rename Variable, Extract Variable* to make the code a bit more readable.
    d. Perhaps try *Extract Class* and *Move Method* to start clarifying the structure of the code.
    e. *When you feel like you understand what the code is doing, turn the page...*

Part b: Prepare for the Omega13!

1. ***Run all the tests.*** At this point, if they don't pass, you cannot proceed. If getting to "Good" will take a lot of effort, start over with a fresh (but stinky) *TestedTrek.*
2. Refactoring is not done merely to make the code pretty. We refactor immediately before or after adding new functionality. When done before, we intend to make it easier to add the new behaviors cleanly. Imagine that you are preparing to implement the *Omega13* user story: You are told only that it functions similarly to the other starship weapons. How would you redesign the existing code so it will be easy to add the Omega13? Refactor with that goal in mind.
    a. You are ***not*** going to be adding the Omega13 (yet!)
    b. You still must ***not*** alter *Untouchables*.
    c. Remember to **run the tests** frequently.

Draw from your pair's/trio's/mob's own experience and knowledge. Also, the resulting code will need to be comprehensible to, and maintainable by, the rest of the developers on your team. E.g., don't use lambda functions if the other devs are not familiar with lambdas. (Alternatively, quickly teach them about lambda functions.)

Optional: Emerging Patterns

What software Design Patterns emerged, were emerging, or you suppose were likely to emerge from your efforts?