Don’t do what I did

Yossi Moalem
Case 1
The scenario

• Low level visualization function
• In edge cases, displayed the object rotated

• Fix the bug
The problem

• Not as rare as I would liked it to be
  • Other parts of the application accommodated for this bug

• Revert it
Sync with everyone and fix at the same time

• Can work on small scale
• Can work if one person can do the work
• Different teams – different priorities
• If one team fails – we should revert the whole fix
Gradual migration

```java
void drawObject ( ... ) {
    // faulty Implementation;
}
```

```java
void drawObjectEx ( ... ) {
    // correct Implementation;
}
```
Two different implementations

```c
void drawObject ( ... ) {
    // faulty Implementation;
}

void drawObjectEx ( ... ) {
    // correct Implementation;
}
```
Call the first implementation:

```c
void drawObject ( ... ) {
    //faulty Implementation;
}
```

```c
void drawObjectEx ( ... ) {
    drawObject( ... );
    //accommodate for the bug
}
```
Call the new one

```c
void drawObject ( ... ) {
    drawObjectEx( .... );
    // Recreate the bug
}

void drawObjectEx ( ... ) {
    // correct Implementation;
}
```
void drawObject ( ... ) {
    drawObjectEx( .... );
    // Recreate the bug
}

void drawObjectEx ( ... ) {
    // correct Implementation;
}
Not only bug fixes - Hyrum's Law

With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody.

- Order the configuration was parsed was changed
- Old order was dependent upon
Document the assumptions

// Order is important

Hi All,

Further to our conversation regarding the configuration parsing order,
Make the assumptions explicit

```cpp
sort (begin(rules), end(rules),
    [](const Rule & lhs, const Rule & rhs )
    { return lhs.priority < rhs.priority; });
```
Takeaways

- Your fix will affect other parts of the application
  - The lower the fix – the more risky it is

- Prepare migration path
  - End with good, clean code

- Document any assumption
  - In code
Case 2
The architecture
The Requirement

Process crash on DB roll back attempt
Possible solutions:

- Move the connection dependent code to DAL
  - Requires process familiarity
- Add observer
  - Very technical solution
Resolving with observer:

• Many more edge
• Needed another observer.
• Well, actually 2 of them

• And it was over
  • It wasn’t

This was obviously the wrong choice!
When to stop:

• Set a limit
  • Time, amount of bugs.

• When this time ends,
  • If possible stop altogether.
  • Or, go back to the scratching board

• And stand by it
  • This is easier said than done….
Takeaways:

• Not knowing the code is not a reason for sub-optimal decisions
  • Never decide unless you DO know the code

• If you keep finding errors in the solution – reconsider it

• Throwing away bad solution and restarting is not a failure
The re-write:

• Too risky!
Case 2
The scenario

- Component with sub-optimal code
  - Many bugs
  - Hard to debug
- Bug in one of the flows

Refactor the code!
The problem

- Each fix called for another fix
- Ended out with very large patch
- Unsafe to introduce
Limiting the refactoring size

• Repeat forever:
  • Find small changes and fix them

Problem:
May spend a lot of time on unimportant fixes
What we really want:

- Introduce many, small changes
- But limit ourselves to the “important” changes
- “peek” at the issues we would encounter
- And fix only those
The incremental way

Master

The fix we want to introduce

Small fix
The incremental way

Master
Takeaways:

• Limit the size of the fix

• Plan for the big fix
  • And push fixes that will help you get there.

• Incomplete fix are fine. As long as they are in the right direction
Design based on technology
rather than need/user
Current architecture

- Server only transfers messages between devices
- Max 20k devices
Requirement:

• Unlimited scalability
• Transparent migration
New architecture

- Device
- Load balancer
- MGMT
- server
- Main MQ
- Backup MQ
- server
- server
- Main Watchdog
- Backup Watchdog
New architecture
New architecture

- devices
- Load balancer
- server
- server
- server
- MGMT
- Main Message queue
- Backup Message queue
- Main Watchdog
- Backup Watchdog
The twist

• A new, urgent requirement came
• Refactor time!
Optimization keys:

Overcommitting
Load Sharing
Simplicity
  Simple containers
  Simple algorithms
Optimize for the rare case
Locking contention
Copy instead of move/share
Memory layout
Takeaways:

• Rally challenge requirements
  • Importance
  • Specific

• Don’t solve bigger problems than what you actually have

• KISS. Always!

• Automation testing worth its weight in gold!
• Thank you!
center the application/ user experience on DB structure
flood with options
The main two points:

- Frame the time spent on fix/feature
  - Back to the scratching board
other

• The phoenix code
• Die fast (threads with pokemon catch)

• Make pattern.
• Not too specific
• How to recognize the pattern. When I encounter this.
  • If a, b and c happens – this is what you do

• Tools, profiling…