# SOFTWARE DESIGN X-RAYS

## Fix Technical Debt With Behavioral Code Analysis **by Adam Tornhill**


### Technical Debt

- Explain the need for refactorings
- Communicate technical trade-offs

Apply at all levels (Micro and Macro)

### Interest Rate Is a Function of Time

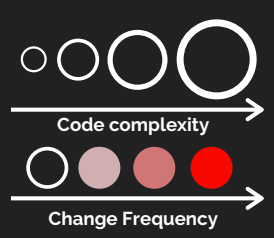
Bad Code is Technical Debt if you have to PAY INTEREST ON IT

## Identify Code with High Interest Rates

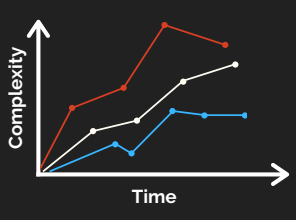
### Prioritize Technical Debt with Hotspots

Complicated code that you have to work with often

- Change frequency of each file
- Lines of code as a simple measure of code complexity




**Hotspot**




### Evaluate Hotspots with Complexity Trends

- Complexity : indentation-based complexity
- Language agnostic

### X-Ray analysis

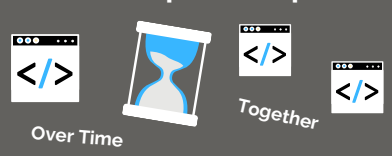
Prioritized list of function to :

- Inspect
- Possibly refactor

## Coupling in Time - A Heuristic for the Concept of Surprise

Change coupling - 2 (or more) files change

- Invisible in the code itself
- Mine it from code's history and evolution




Is and Isn't Temporal Coupling (ex : Unit Tests)

Neither good nor bad all depends on context

"Change coupling can help us design better software as we uncover expensive change patterns in our code"

## Refactor Congested Code with the Splinter Pattern

Break a hotspot into smaller parts

- Along its responsibilities
- Maintaining the original API for a transient period

"Parallel Development Is at Conflict with Refactoring"

### How to ?

1. Ensure tests cover the splinter candidate
2. Identify the behaviors inside your hotspot
3. Refactor for proximity
4. Extract a new module for the behavior with the most development activity
5. Delegate to the new module
6. Perform regression tests
7. Select the next behavior to refactor and start over at 4

## Stabilize Code by Age

- Organize our code by its age
- Turn stable packages into libraries
- Move and refactor code we fail to stabilize

- Promotes long-term memory models of code
- Less cognitive load : less active code
- Prioritizes test suites to shorten lead times

"Always remember that just because some code is a hotspot, that doesn't necessarily mean it's a problem."

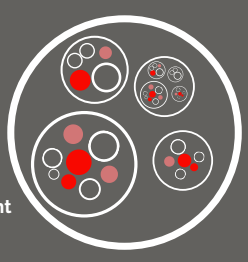
## Divide and Conquer with Architectural Hotspots

### Identify your architectural boundaries :

Often based on the folder structure of the codebase

### Hotspot analysis on an architectural level :

- Identify the subsystems with the most development effort
- Visualize the complexity trend of a whole architectural component



### Analyze the files in each architectural hotspot

### Fight the Normalization of Deviance

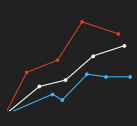
- Each time you accept a risk, the deviations become the new normal
- Complexity trends as WHISTLEBLOWERS

"The more often something is changed the more important it is that the corresponding code is of high quality so all those changes are simple and low risk"

## Communicate with Nontechnical Managers - Data buys trust


### % of commits involving top hotspots

- Demonstrate importance of this code
- Support new features and innovations



### Show complexity trends

- Code gets worse over time
- Which will slow us down

### Coordination bottlenecks

- Add people side to the presentation

## Beyond Conway's Law

### Quality Suffers with Parallel Development

Increases risk of defects with the number of developers

### Coordination needs

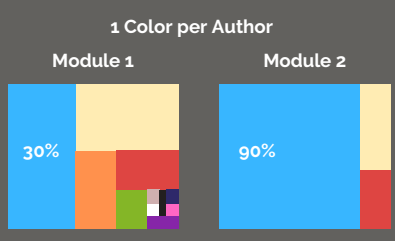
Number of authors behind each component

### Rank Code by Diffusion


### Calculate a fractal value

- How many different authors have contributed
- How the work is distributed among them

0 : Single author
1 : the more contributors there are




### Module 1 : Many minor contributors

Higher risk for defects


### Module 2 : 1 main developer

Reduced risks

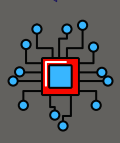
"Ranks all the modules in our codebase based on how diffused the development effort is"

### Use Fractal Values to

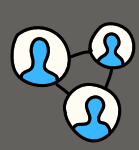
**Prioritize code reviews**
Done right = a proven defect-removal

**Focus tests**
Identify the areas to focus extra tests

**Replan suggested features**
If high developer congestion



**Redesign for increased parallelism**
Candidate for splinter refactorings ?



**Introduce areas of responsibility**
introduce teams aligned with the structure of the code

## Use Social Data

### Fight motivation losses in Teams

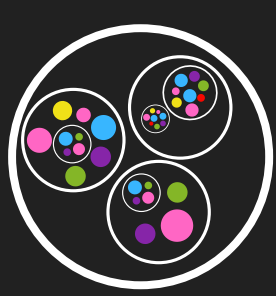
**Evaluation**
Someone else cares about your contribution

**Lead by example**
Model the behaviors you want to see in others


**Visibility**

- Recognize contributions
- Present knowledge maps

**Small Groups**



**Knowledge Map**
Main Author / Module

### Guide On and Off-boarding

**Identify the Experts**
Find out who to communicate with

**Measure Future Knowledge Loss**

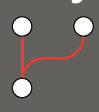
**React to Knowledge Loss**
Focus to maintain knowledge

## Biases and Workarounds for Behavioral Code Analysis

**Data**
Minimum amount of data

**Incorrect author info**
Need a minimum amount of data

**Copy-paste repositories**
Fails to migrate its history



**Misused squash commits**
When applied to work committed by several individuals

**Discover it in action on codescene.com** **#sharingiscaring** **by Yoan THIRION** **@yot88**