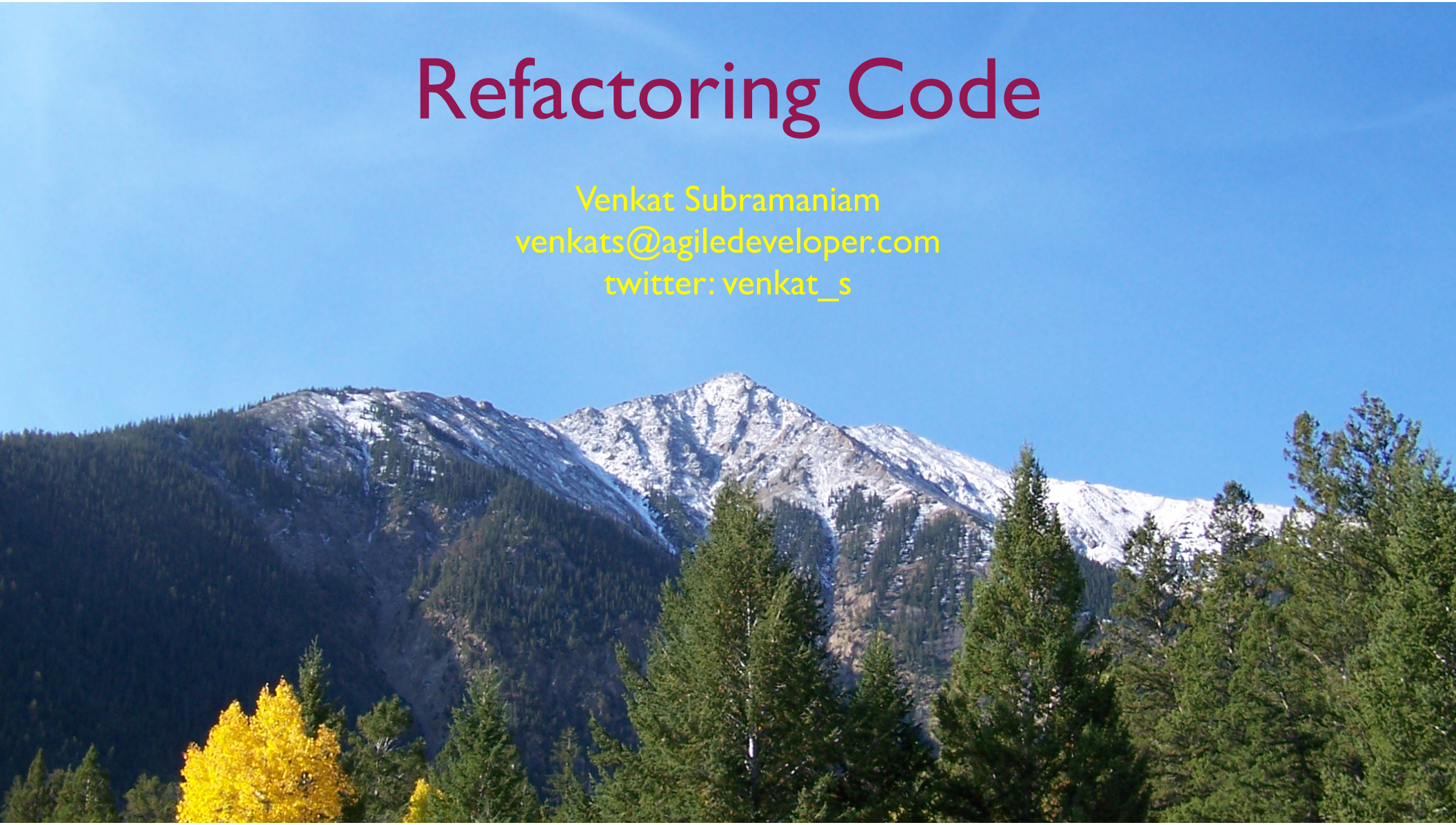
Refactoring Code

Venkat Subramaniam
venkats@agiledeveloper.com
twitter: venkat_s



What's & What-nots

In this presentation

We'll cover

Why refactor?

When?

How?

Principles/Practices to follow

What's Refactoring?

Your genuine desire to improve the quality of your code and design in it

But Why?

You Can't be Agile if your code sucks!

But, It Takes Time

Yes, it will take time

Mind Your Speed

"Lowering quality lengthens development time"—First Law of Programming.

Why Refactor?

To make the code easier to understand

To make it easier to maintain

To make change affordable

After all "Change is the only constant"
—Confucius

It helps you prepare to "Embrace Change"

Why Refactor?

"Programs must be written for people to read, and only incidentally for machines to execute"—Abelson and Sussman.

Strive to Evolve

You can't write perfect code in one sitting—impossible

Design, rather than happening right just once, evolves continuously during development

Code that's hard to understand is worst than code that's lost

Evolve It

Make it work first, then make it better

Benefit

Refactoring reduces your risk—can lead to lightweight pragmatic design

What's Refactoring Again?

"Art of improving the design of existing code"

"A 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"—Martin Fowler in his Refactoring book

Strike A Balance

Just because you think you need to change, it does not mean it needs change

Consider cost and impact of change

Set a second opinion

Don’t soldier alone

Refactoring is Hard?

It can be

Like everything else in life—driving, speaking, socializing,...

It depends on how we approach it

Shalt Not Fear Change

"The only thing to fear is fear itself"—
FDR.

Why Fear Refactoring?

What if I break something that worked?

Is my change worst than the original code?

We hate being embarrassed, it's easy to leave things as is

Some Principles

Let's consider some principles that can help Refactoring

Zeroth Principle

Rely on automated tests

Most ideal if you can have unit tests

If you can't, high level functional/ integration test is good

Isolate candidate code and create test if you have to

What to Look For?

Surprisingly, real good advice comes from an old book on writing good English!

On Writing Well

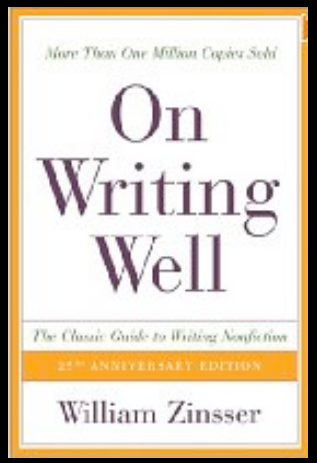
William Zinsser on writing non-fiction

Simplicity

Clarity

Brevity

Humanity



First Principle

Reduce code

Don’t write code that’s really not needed

Programmers write as much code as restaurants serve food—way too much

Code you don’t write, don’t have to be maintained!

Attain True Perfection

"Perfection is achieved, not when there is nothing left to add, but when there is nothing left to remove"—Antoine de Saint-Exupery

Second Principle

Avoid Clever Code—Keep it Simple

Make it clear, not clever

Third Principle

Make it small and cohesive

Small and Cohesive

Avoid long methods

Assign single responsibility to each method and each class

If it does not belong here, don't add it

Fourth Principle

Eliminate Duplication

Keep code DRY

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system"—Andy Hunt and Dave Thomas, in The Pragmatic Programmers

Fifth Principle

Eliminate Dependency

Don’t strive to reduce dependency/ decoupling

Get rid of it

Decouple only when you can’t eliminate

Sixth Principle

Make comments redundant and remove them

Make code self documented

Write executable comments: A good test is worth a thousand comments

Seventh Principle

Make sense in seconds, not in minutes, hours, weeks, ...

If you have to read through every line of code and think you lost it

It's not just about size, its about conveying intent explicitly

Time to Understand?

```
public static void exitIfCheckinIncludesSelectFiles(String changes)
{
  for(String line : changes.split("\n"))
  {
      if (line.startsWith("A") || line.startsWith("U"))
      {
        if (line.endsWith("Debug/") ||
            line.endsWith("bin/") ||
            line.endsWith(".class") ||
            line.endsWith(".exe"))
        {
            printErrorMessageAndExit(MESSAGE);
        }
      }
  }
}
```

How about this version?

```
public static void exitIfCheckinIncludesSelectFiles(String changes)
{
  for(String line : changes.split("\n"))
  {
      if (line.startsWith("A") || line.startsWith("U"))
      {
        if (line.endsWith("Debug/") ||
            line.endsWith("bin/") ||
            line.endsWith(".class") ||
            line.endsWith(".exe"))
        {
            printErrorMessageAndExit(MESSAGE);
        }
      }
  }
}
```

```
for(String line : changes.split("\n"))
{
    if (ifFileIsInvalid(line)) printErrorMessageAndExit(MESSAGE);
}
```

Eighth Principle

Avoid Primitive Obsession

Avoid desire to operate at lowest level

Instead use, look for, or create higher level easy to use abstraction

Primitive Obsession

```
def isSPellingCorret(word) {

  File file = new File("...")

  def found = false
  file.eachLine {
    if (it == word) found = true
  }

  found
}
```

Removing Obsession

```
def isSPellingCorret(word) {

  File file = new File("...")

  file.readLines().contains(word)
}
```

Ninth Principle

Checkin Frequently, take small steps

Frequent Checkin

Don't hold code for extended period of time

Merge becomes painful

If you lock out others, you inhibit their progress

Big bang integration is a big bang fail

By checking in frequently, you allow for short quick feedback cycle

Your changes are relevant, exercised, and validated right away

Tenth Principle

Keep code at one level of abstraction

Compose Method where each method addresses one level of abstraction

Refactoring Opportunity?

How do you know which code needs refactoring?

General awareness to sense smelly code

Use tactical code reviews

Make refactoring a regular activity, each day

When Not to Refactor

Code is Messed up Beyond Any Possible Repair

When you're in the middle of fixing a bug

When in middle of another change or refactoring

- Make a note to visit later

If you don't see clear benefit to the particular refactoring activity

When to Refactor?

Before fixing a bug

After fixing a bug

Before a design enhancement

After a design enhancement

If you think you will improve quality of code/design

If you can make it easier to understand

How to Refactor?

Small steps—devise sequence of small steps to take

Be continuous, not episodic

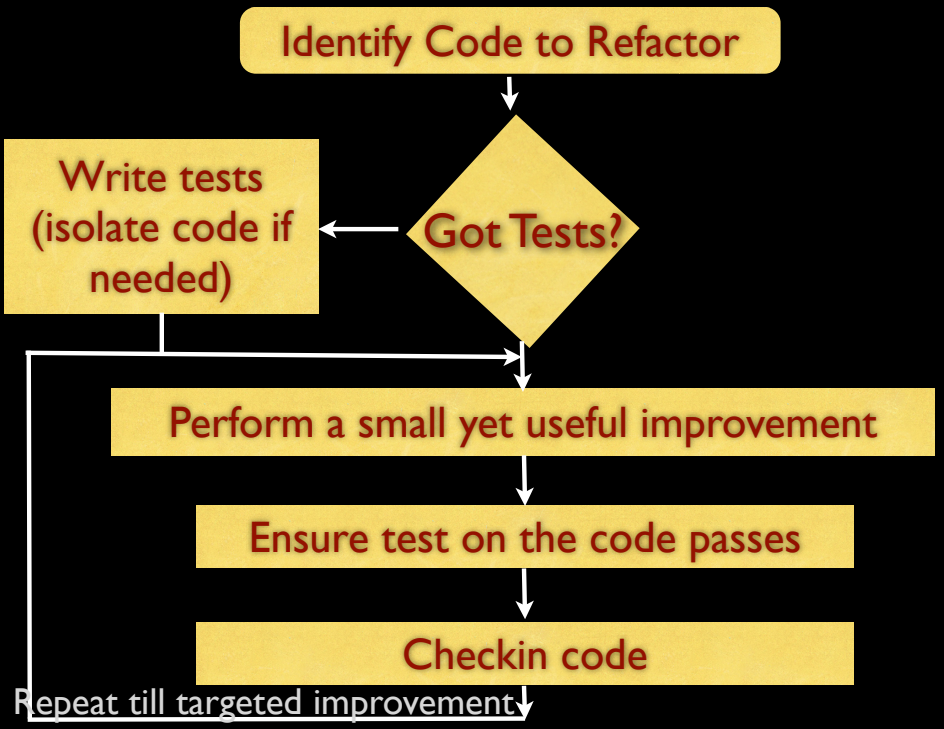
Aim for bite-size improvements

Never refactor code that's not in version control

Don't hesitate to throw out change

Check in frequently (every few minutes)

The Flow


Identify Code to Refactor
Got Tests?
Write tests (isolate code if needed)
Perform a small yet useful improvement
Ensure test on the code passes
Checkin code
Repeat till targeted improvement


Thank You!
Venkat Subramaniam
venkats@agiledeveloper.com
twitter: venkat_s