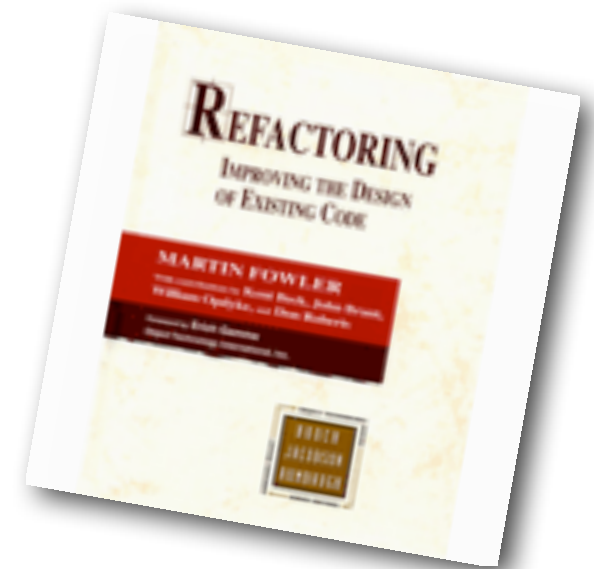


Refactoring

“Design is too important to be done only when we know nothing about the project”

Refactoring

- *Refactoring* is improving the design of existing code
- Two choices:
 - Design up front, getting everything exactly right the first time, or
 - Design as you go, and be prepared to refactor

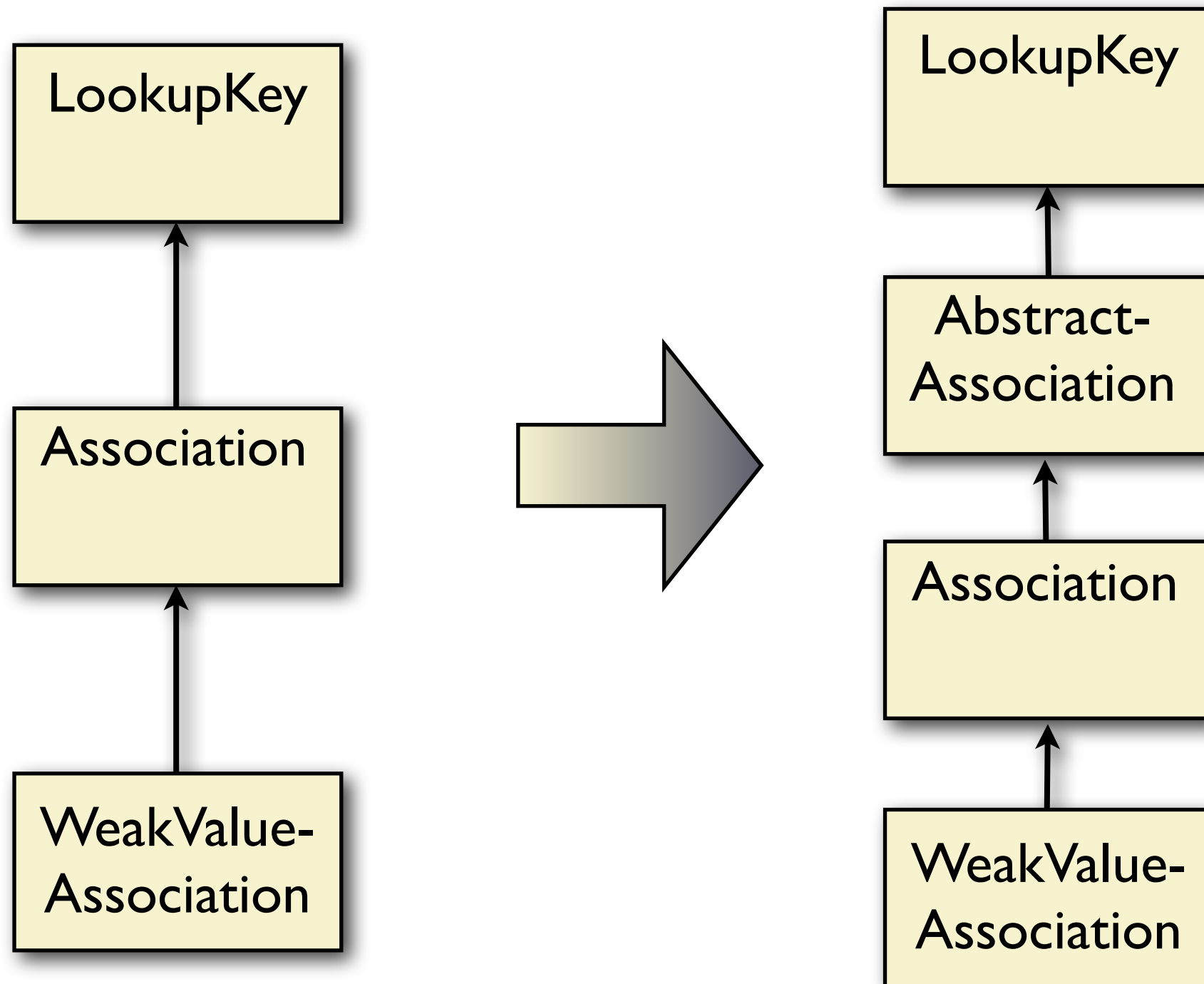


These slides based on materials by Don Roberts and John Brant

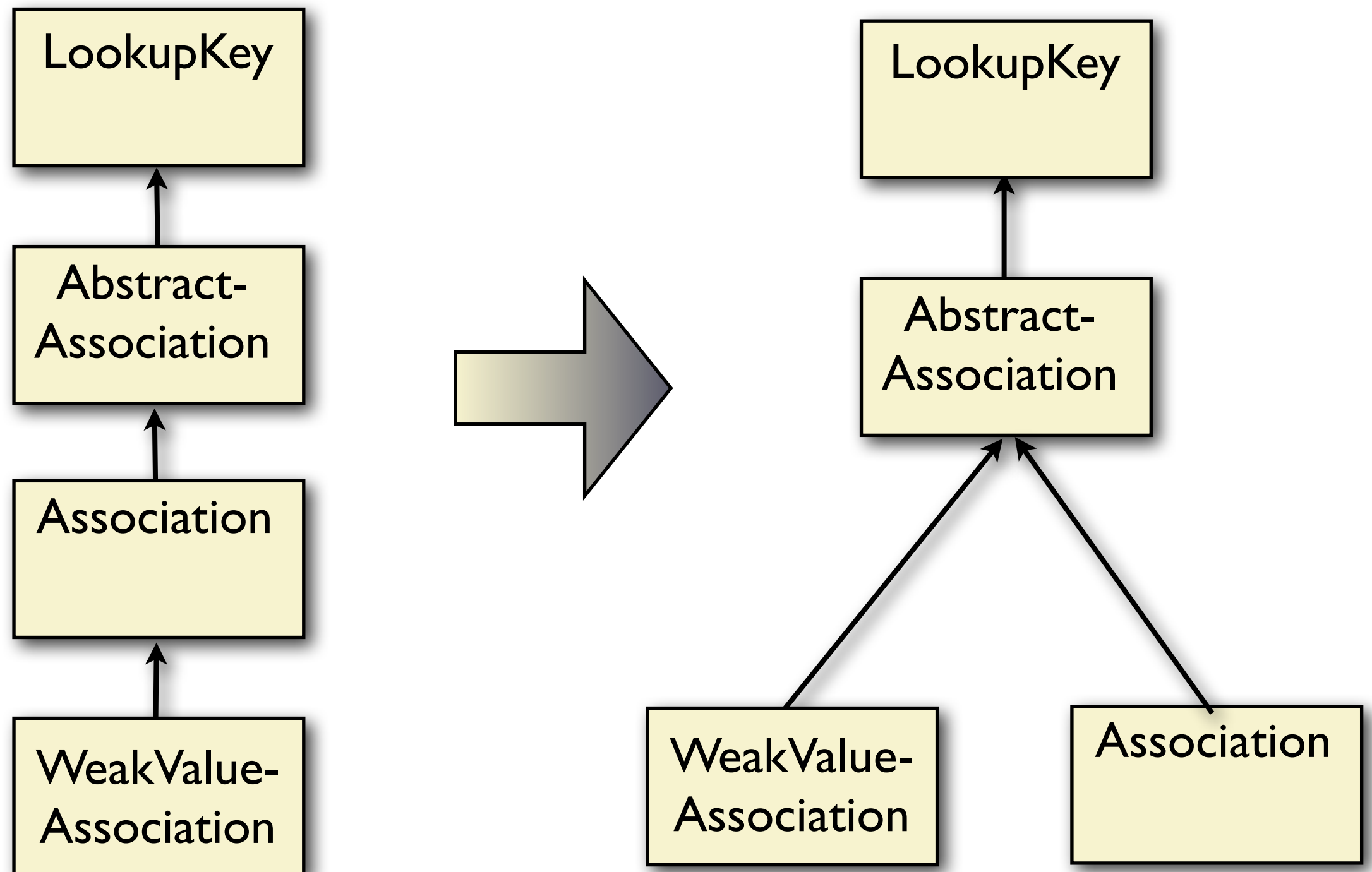
Software Maintenance

- Practically all “maintenance” is just continuing development
- Initial development is just “maintaining” a blank sheet of paper
- Software is never finished...
 - until it’s pried from the cold dead hands of its last user.

A Simple Refactoring: Add Empty Class



Another Refactoring ...



So, what's the problem?

- Complexity
 - It's hard to understand what's there
- Fear
 - Changing what you don't understand is scary
- Errors
 - If you get it wrong, you break a working program

If it ain't broke, don't fix it.

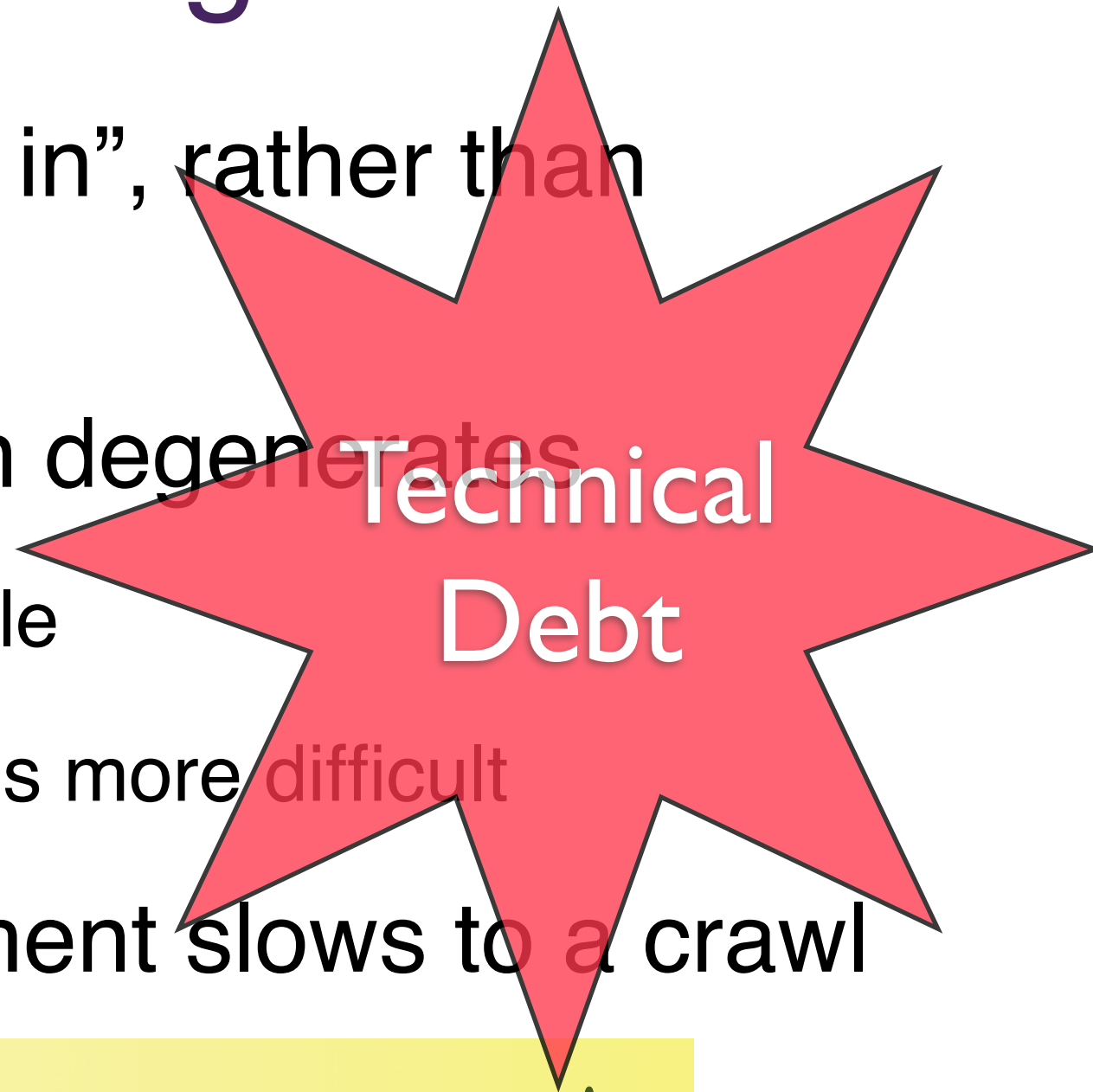
Schedule Pressure

- Every project is in a time crunch
- Refactoring can be time consuming
 - ▶ wouldn't it be better to put it off until after the next release?
- You are being paid to add *new* functionality

If it ain't broke, don't fix it.

Consequences of deferring refactoring

- Changes are “hacked in”, rather than designed
- Overall system design degenerates
 - Code becomes more brittle
 - The next change becomes more difficult
- The pace of development slows to a crawl



Don't let this happen to your system!

The Refactoring Process

- Think about manipulating a mathematical expression:

$$ax^2 + bx + c \rightsquigarrow axx + bx + c \rightsquigarrow (ax + b)x + c$$

- Each step is semantics-preserving, so many small steps can be combined to have a large effect

Individual Refactorings

- Add Something:

- Add field
- Add temporary variable
- Add Class variable
- Add Class
- Add methods

- Remove Something:

- Remove field
- Remove temporary variable
- Remove Class variable
- Remove Class
- Remove methods

- Rename Something:

- Rename field
- Rename temporary variable
- Rename Class variable
- Rename Class
- Rename methods (see next slide)

- Move Something:

- Move field up or down
- Move temp to inner/outer scope
- Move class variable up or down
- Move method to component
- Move field to component
- Change superclass

Method-level Refactorings

- Method Renamings

- Simple rename

- Permute arguments

- Add argument

- Remove argument

- Introductions

- Extract code into method

- Extract code into temporary variable

- Eliminations

- Inline method

- Inline temporary

Safe Refactoring

- Use tests
 - tests should pass before and after refactoring
- Use a refactoring tool if it's available
 - Smalltalk Refactoring Browser
 - Plugins for Java in Eclipse
- Take small steps, testing between each step

Code Smells

- Develop a “nose” for code
 - Does the code smell bad?
- What bad smells have you seen in others' code?

Some smells
that I have known

Code violates the “once and only once” rule

- code does not say it *at all*
- code says it *twice, thrice, ... fifteen times!*

Methods are too large

- Why is this a problem?
 - ▶ methods are the smallest unit of overriding
 - ▶ statements in a method should be at same level of abstraction

Methods in the wrong class

- if a method does not refer to **self**, it is probably in the wrong class
 - implicit **self** counts as referring to self
- check the parameters
- However:
 - there are “utility methods” that have no natural home

“Feature Envy”

- method over-uses accessors (getters and setters) of another object
- can the method be moved into the other object?
 - ▶ sometimes only *part* of the method should be moved
 - ▶ extract method into component

The “God” class

- a large class with many methods and many fields
- can you partition the methods and the fields that they access?
- turn each partition into new class
 - large class becomes composition of smaller classes

Field not always used

- Some instances use a particular field, others don't
- Create two or more subclasses with the *right* fields
- Is a field used only during a certain operation?
 - “operation” spans more than one method
 - consider using a method object

Co-occurring Parameters

- if the same pair (or triplet) of parameters is passed to several methods:
- perhaps they represent an abstraction that should be captured in an object?
 - ▶ e.g., *x* and *y* should be grouped into a point object
 - ▶ e.g., *list* and *index* should be grouped into an iterator object
- once the object exists, you will often find it natural to add *behavior*

Comments

- Most comments are written to compensate for poorly written code!
 - ▶ if you feel that your code needs *explaining*, consider *refactoring* it instead

Original Code

initialize

I w button I

super initialize.

self layoutPolicy: TableLayout new.

self listDirection: #leftToRight.

self layoutInset: 2.

self borderWidth: 0.

self hResizing: #shrinkWrap.

self vResizing: #shrinkWrap.

self color: Color gray.

w := TheWorldMenu new

world: World project:

(World project ifNil: [Project current])

hand: World primaryHand.

button := LaunchButtonMorph new.

button label: 'Browser';

actionSelector: #openBrowser;

target: Browser;

actWhen: #buttonUp.

self addMorph: button.

button := LaunchButtonMorph new.

button label: 'Workspace';

actionSelector: #openWorkspace;

target: w;

actWhen: #buttonUp.

self addMorph: button.

button := LaunchButtonMorph new.

button label: 'Transcript';

actionSelector: #openTranscript;

target: w;

actWhen: #buttonUp.

self addMorph: button.

button := LaunchButtonMorph new.

button label: 'Change Sorter';

actionSelector: #openChangeSorter2;

target: w;

actWhen: #buttonUp.

self addMorph: button.

button := LaunchButtonMorph new.

button label: 'File List';

actionSelector: #openFileList;

target: w;

actWhen: #buttonUp.

self addMorph: button

Add comments and explaining names

initialize

```
I w browserButton workspaceButton  
transcriptButton changeButton fileListButton I
```

```
super initialize.
```

"initialize layout"

```
self layoutPolicy: TableLayout new.  
self listDirection: #leftToRight.  
self layoutInset: 2.  
self borderWidth: 0.  
self hResizing: #shrinkWrap.  
self vResizing: #shrinkWrap.  
self color: Color gray.
```

```
w := TheWorldMenu new
```

```
world: World project:  
    (World project ifNil: [Project current])  
hand: World primaryHand.
```

"initialize buttons"

```
browserButton := LaunchButtonMorph new.  
browserButton label: 'Browser';  
actionSelector: #openBrowser;  
target: Browser;  
actWhen: #buttonUp.  
self addMorph: browserButton.
```

```
workspaceButton := LaunchButtonMorph new.  
workspaceButton label: 'Workspace';  
actionSelector: #openWorkspace;  
target: w;  
actWhen: #buttonUp.  
self addMorph: workspaceButton.
```

```
transcriptButton := LaunchButtonMorph new.  
transcriptButton label: 'Transcript';  
actionSelector: #openTranscript;  
target: w;  
actWhen: #buttonUp.  
self addMorph: transcriptButton.
```

```
changeButton := LaunchButtonMorph new.  
changeButton label: 'Change Sorter';  
actionSelector: #openChangeSorter2;  
target: w;  
actWhen: #buttonUp.  
self addMorph: changeButton.
```

```
fileListButton := LaunchButtonMorph new.  
fileListButton label: 'File List';  
actionSelector: #openFileList;  
target: w;  
actWhen: #buttonUp.  
self addMorph: fileListButton
```


Composed Method

initialize

```
super initialize.  
self initializeLayout.  
self initializeButtons
```

initializeButtons

```
| w |  
w := TheWorldMenu new  
    world: World  
    project: (World project ifNil: [Project current])  
    hand: World primaryHand.  
self addAButton: 'Browser' sending: #openBrowser to: Browser.  
self addAButton: 'Workspace' sending: #openWorkspace to: w.  
self addAButton: 'Transcript' sending: #openTranscript to: w.  
self addAButton: 'Change Sorter' sending: #openChangeSorter2 to: w.  
self addAButton: 'File List' sending: #openFileList to: w
```

initializeLayout

```
self layoutPolicy: TableLayout new.  
self listDirection: #leftToRight.  
self layoutInset: 2.  
self borderWidth: 0.  
self hResizing: #shrinkWrap.  
self vResizing: #shrinkWrap.  
self color: Color gray.
```

addAButton: label sending: s to: target

```
| button |  
button := LaunchButtonMorph new.  
button label: label;  
    actionSelector: s;  
    target: target;  
    actWhen: #buttonUp.  
self addMorph: button
```

Nested Conditionals

- Message send = procedure call + case selection
 - use this to eliminate explicit explicit conditionals
 - the goal: adding new cases does not require changing existing code
- e.g., instead of testing isEmpty or isNil, consider a separate object to represent the Empty or Nil case
 - The Null Object Pattern (<http://www.cs.oberlin.edu/~jwalker/refs/woolf.ps>)

Nested Conditionals (cont.)

- Early returns are often better than nested conditionals.

Text>>emphasisAt: characterIndex

"Answer the font for characters in the run beginning at characterIndex."

| attributes emph |

self size = 0 ifTrue: [↑ 0].

emph := 0.

attributes := runs at: characterIndex.

attributes do: [:att | emph := emph bitOr: att emphasisCode].

↑ emph

- Is there a need for the test at all?

Nested Conditionals (cont.)

- If conditional involves a test of the object's class, move the method to that class
 - ▶ self class = ... or
 - ▶ isKindOf:

AbstractSound >>loudness: aNumber

"Initialize my volume envelopes and initial volume. ..."

I vol I




vol := (aNumber asFloat max: 0.0) min: 1.0.

envelopes do: [:e I (e isKindOf: VolumeEnvelope) ifTrue: [e scale: vol]].
self initialVolume: vol.

Strategies for Refactoring


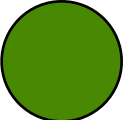
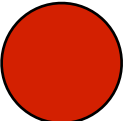

1. Extend then Refactor
2. Refactor then Extend
3. Debug then Refactor
4. Refactor then Debug
5. Refactor for Understandability

Extend then Refactor

- test fails 
- hack in a change to make the test pass 
 - e.g., copy and paste a method, and then edit the new method.
- test passes, but you are not done yet! 
 - eliminate redundancy

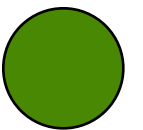
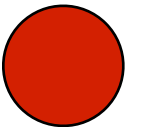
**Coding is like mountain climbing: getting
the green light is like reaching the summit**

Refactor then Extend

- It seems awkward to implement a new feature 
- Refactor design to make the change easy 
- add a test for the feature 
- add the feature 

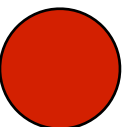
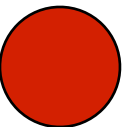
Debug then Refactor

- Find the bug
- Fix the bug
- Refactor to make the bug obvious, e.g.,
 - extract method and give it an *explaining name*
 - rename method or temp
 - extract expression to temporary variable
 - eliminate “magic numbers”



Refactor then Debug

- Suppose that you can't find the bug?
 - Refactoring preserves bad behavior too!
- Simplify complex method
- Fix the bug



Refactor for Understandability

- What was obvious when a method was written isn't always obvious a day later!
 - ▶ use composed method (Beck p. 21)
 - ▶ use intention revealing selectors (Beck p. 49)
 - ▶ use explaining temporary variable (Beck p. 108)
 - ▶ don't worry about performance
 - “clever” code is usually dumb

The Loan Metaphor

“Quick and dirty” coding is like taking out a loan

Living with the bad code is like paying interest

Refactoring your code is like paying off the loan

— — — —

- Some debt is OK, in fact necessary, to grow a business
- Too much debt is unhealthy: it will eventually kill you

**“Technical Debt”
must be paid off**

Listen to your Code

- If something seems difficult or awkward, refactor to make it easy
- Let the program tell you where it needs to be fixed
 - Does the code speak to you? Does it smell?
- If you copy and paste, you *probably want to* refactor to remove the duplication

Do you know all of the refactorings?

List of Refactorings

Add Parameter	275
Change Bidirectional Association to Unidirectional	200
Change Reference to Value	183
Change Unidirectional Association to Bidirectional	197
Change Value to Reference	179
Collapse Hierarchy	344
Consolidate Conditional Expression	240
Consolidate Duplicate Conditional Fragments	243
Convert Procedural Design to Objects	368
Decompose Conditional	238
Duplicate Observed Data	189
Encapsulate Collection	208
Encapsulate Downcast	308
Encapsulate Field	206
Extract Class	149
Extract Hierarchy	375
Extract Interface	341
Extract Method	110
Extract Subclass	330
Extract Superclass	336
Form Template Method	345
Hide Delegate	157
Hide Method	303
Inline Class	154
Inline Method	117
Inline Temp	119
Introduce Assertion	267
Introduce Explaining Variable	124
Introduce Foreign Method	162
Introduce Local Extension	164
Introduce Null Object	260
Introduce Parameter Object	295
Move Field	146
Move Method	142
Parameterize Method	283
Preserve Whole Object	288

Pull Up Constructor Body	325
Pull Up Field	320
Pull Up Method	322
Push Down Field	329
Push Down Method	328
Remove Assignments to Parameters	131
Remove Control Flag	245
Remove Middle Man	160
Remove Parameter	277
Remove Setting Method	300
Rename Method	273
Replace Array with Object	186
Replace Conditional with Polymorphism	255
Replace Constructor with Factory Method	304
Replace Data Value with Object	175
Replace Delegation with Inheritance	355
Replace Error Code with Exception	310
Replace Exception with Test	315
Replace Inheritance with Delegation	352
Replace Magic Number with Symbolic Constant	204
Replace Method with Method Object	135
Replace Nested Conditional with Guard Clauses	250
Replace Parameter with Explicit Methods	285
Replace Parameter with Method	292
Replace Record with Data Class	217
Replace Subclass with Fields	232
Replace Temp with Query	120
Replace Type Code with Class	218
Replace Type Code with State/Strategy	227
Replace Type Code with Subclasses	223
Self Encapsulate Field	171
Separate Domain from Presentation	370
Separate Query from Modifier	279
Split Temporary Variable	128
Substitute Algorithm	139
Tease Apart Inheritance	362

List of Refactorings

Add Parameter	275
Change Bidirectional Association to Unidirectional	200
Change Reference to Value	183
Change Unidirectional Association to Bidirectional	197
Change Value to Reference	179
Collapse Hierarchy	344
Consolidate Conditional Expression	240
Consolidate Duplicate Conditional Fragments	243
Convert Procedural Design to Objects	368
Decompose Conditional	238
Duplicate Observed Data	189
Encapsulate Collection	208
Encapsulate Downcast	308
Encapsulate Field	206
Extract Class	149
Extract Hierarchy	375
Extract Interface	341
Extract Method	110
Extract Subclass	330
Extract Superclass	336
Form Template Method	345
Hide Delegate	157

Extract Hierarchy	375
Extract Interface	341
Extract Method	110
Extract Subclass	330
Extract Superclass	336
Form Template Method	345
Hide Delegate	157
Hide Method	303
Inline Class	154
Inline Method	117
Inline Temp	119
Introduce Assertion	267
Introduce Explaining Variable	124
Introduce Foreign Method	162
Introduce Local Extension	164
Introduce Null Object	260
Introduce Parameter Object	295
Move Field	146
Move Method	142
Parameterize Method	283
Preserve Whole Object	288

Pull Up Constructor Body	325
Pull Up Field	320
Pull Up Method	322
Push Down Field	329
Push Down Method	328
Remove Assignments to Parameters	131
Remove Control Flag	245
Remove Middle Man	160
Remove Parameter	277
Remove Setting Method	300
Rename Method	273
Replace Array with Object	186
Replace Conditional with Polymorphism	255
Replace Constructor with Factory Method	304
Replace Data Value with Object	175
Replace Delegation with Inheritance	355
Replace Error Code with Exception	310
Replace Exception with Test	315
Replace Inheritance with Delegation	352
Replace Magic Number with Symbolic Constant	204
Replace Method with Method Object	135
Replace Nested Conditional with Guard Clauses	250
Replace Parameter with Explicit Methods	285

Replace Constructor with Factory Method	304
Replace Data Value with Object	175
Replace Delegation with Inheritance	355
Replace Error Code with Exception	310
Replace Exception with Test	315
Replace Inheritance with Delegation	352
Replace Magic Number with Symbolic Constant	204
Replace Method with Method Object	135
Replace Nested Conditional with Guard Clauses	250
Replace Parameter with Explicit Methods	285
Replace Parameter with Method	292
Replace Record with Data Class	217
Replace Subclass with Fields	232
Replace Temp with Query	120
Replace Type Code with Class	218
Replace Type Code with State/Strategy	227
Replace Type Code with Subclasses	223
Self Encapsulate Field	171
Separate Domain from Presentation	370
Separate Query from Modifier	279
Split Temporary Variable	128
Substitute Algorithm	139
Tease Apart Inheritance	362