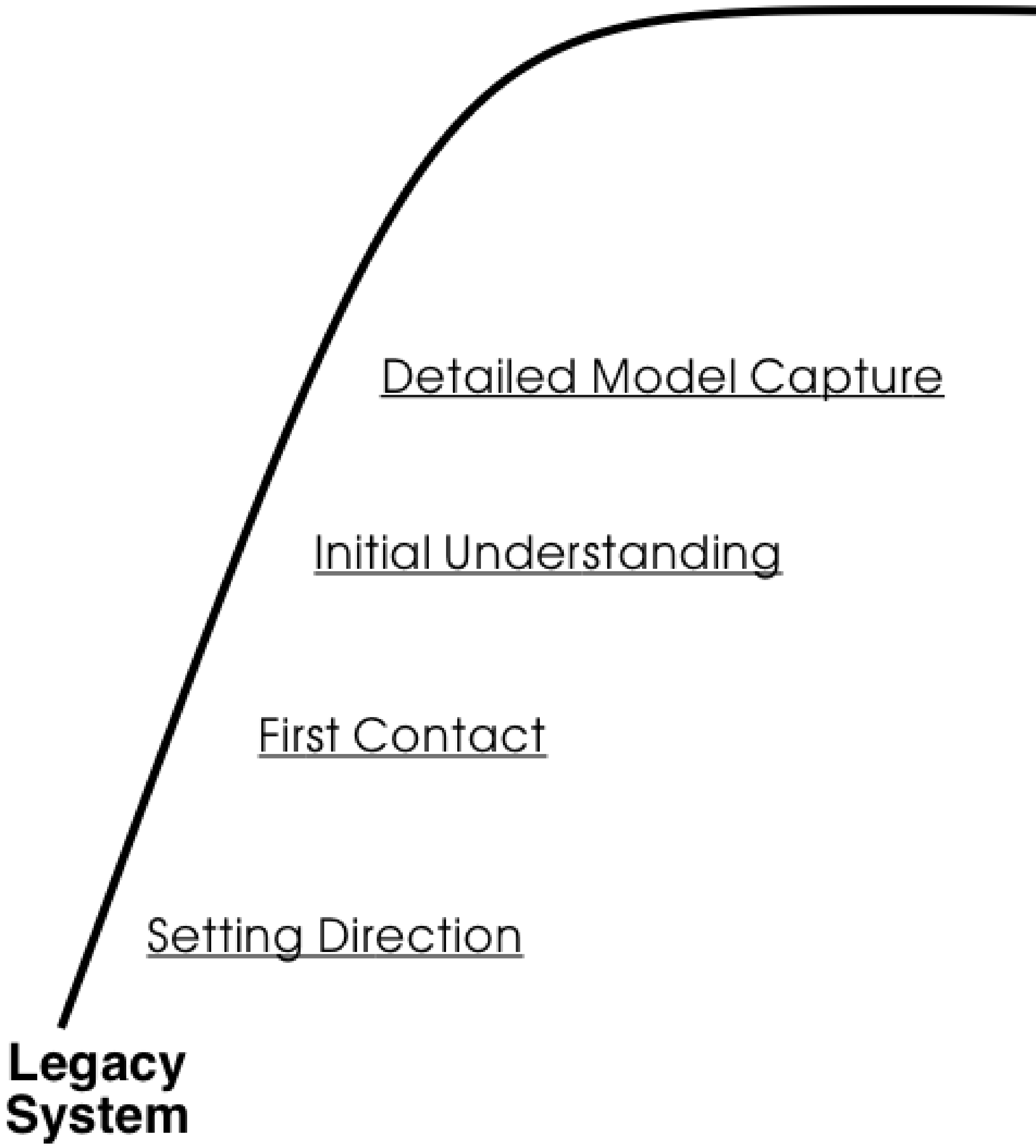
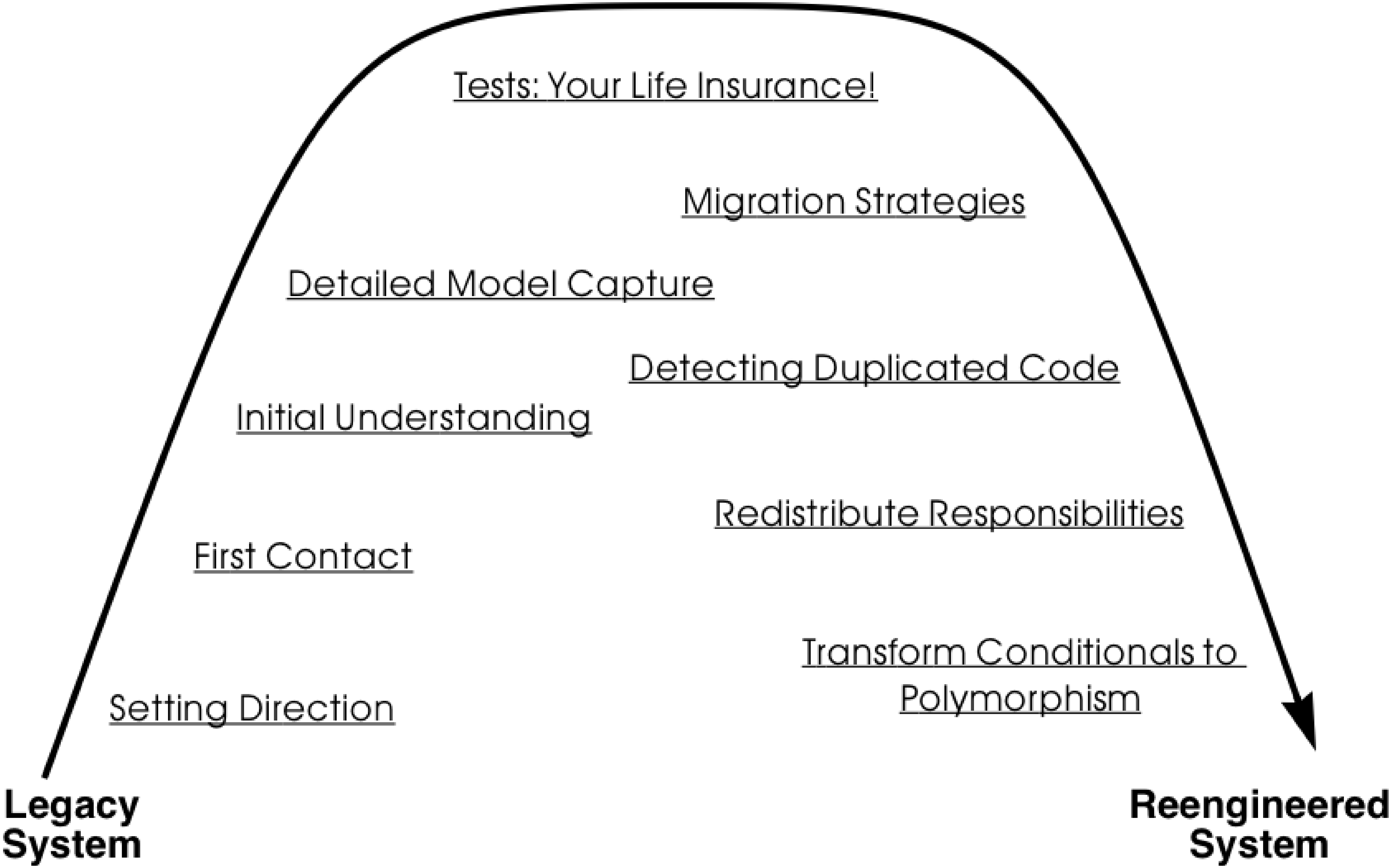
Reengineering II

Transforming the System

Recap: Reverse Engineering

- We have a detailed impression of the current state
- We identified the important parts
- We identified reengineering opportunities
- We have a detailed understanding of the system
- We documented the knowledge of the reverse engineering part





Legacy System

Setting Direction

First Contact

Initial Understanding

Detailed Model Capture

Tests: Your Life Insurance!

Migration Strategies

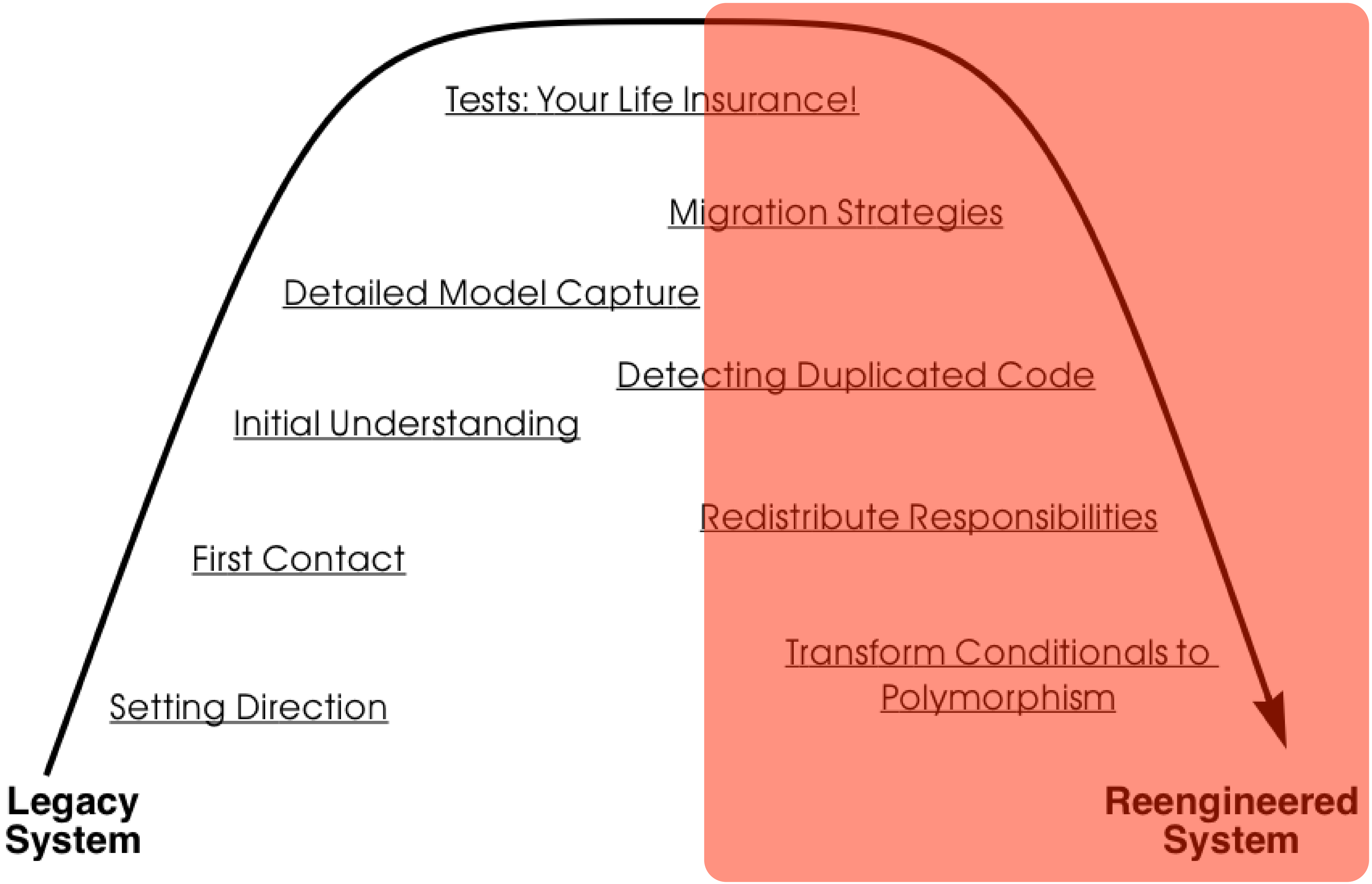
Detecting Duplicated Code

Redistribute Responsibilities

Transform Conditionals to Polymorphism

Reengineered System

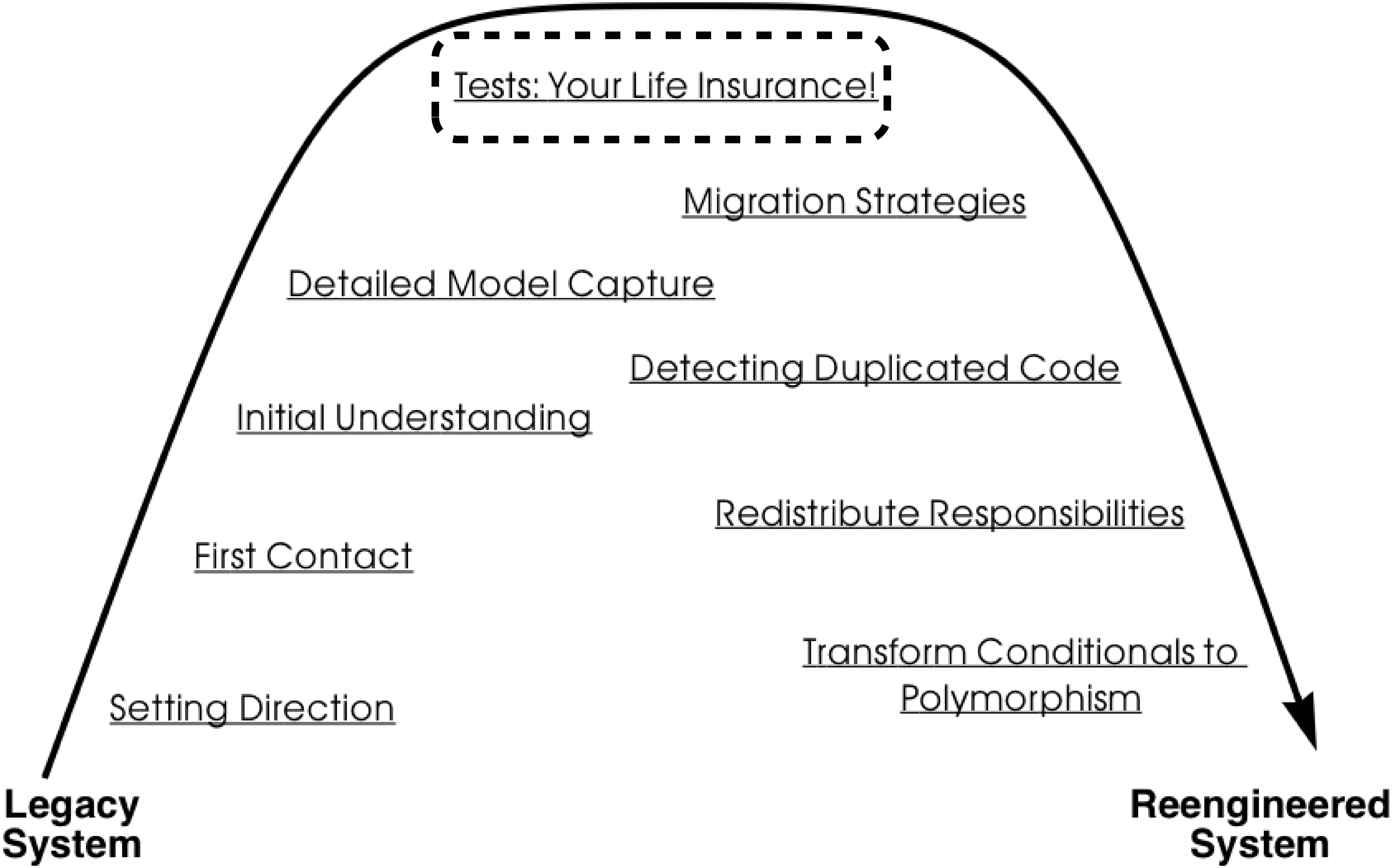
Reengineering Patterns



Reengineering Patterns

Why do we reengineer?

- We want to improve our system in some or the other way
- Improve internal structure of the software
    - Make maintenance “easier”
    - Make new feature implementation “easier”
- Improving technology
    - Game engine: Use new Direct3D API to support latest shaders; multi platform ready: Support OpenGL (PS4, WII U) & Direct3D (XBox, PC)
    - Using object-oriented database mapper, rather than accessing “plain SQL”
- Improve performance
    - Reengineering data model to speed up database queries



Tests: Your Life Insurance!

Migration Strategies

Detailed Model Capture

Detecting Duplicated Code

Initial Understanding

Redistribute Responsibilities

First Contact

Transform Conditionals to Polymorphism

Setting Direction

Legacy System

Reengineered System

Tests: Your Life Insurance! (Ch. 6)

Tests: Your Life Insurance! (Ch. 6)

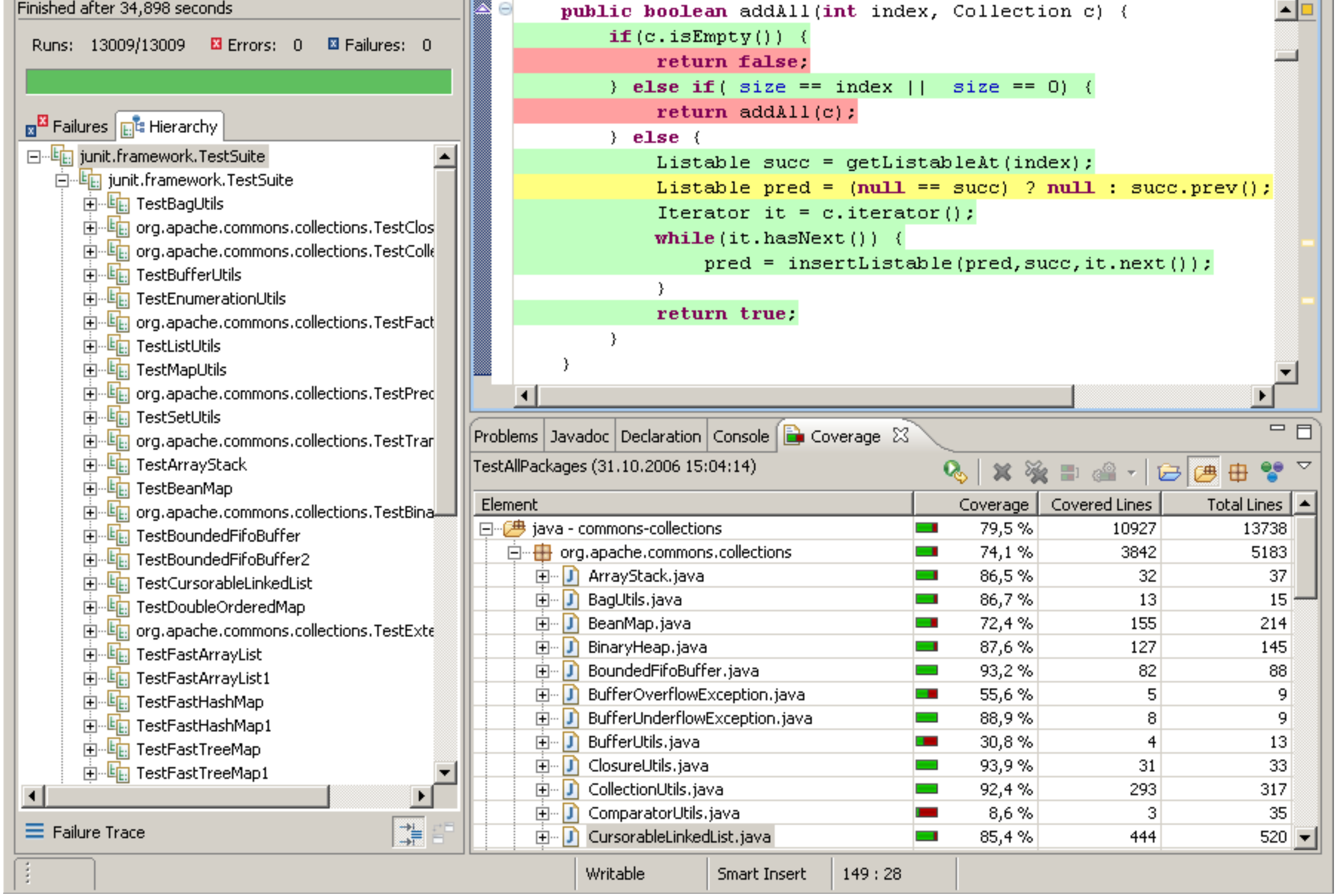
- Reengineering: *Radical surgery* on the (most) *valuable parts* of the system
- For sure, we don not want to introduce new defects or even break any working parts
- Reengineering per se is a risky business with many opportunities to fail
- Unit test can reduce the risks posed by reengineering
- Whenever we change code, we must make use of unit tests

The Problem with Tests

- To write tests, we sometimes need to change the code
- Tests are time consuming: Under time pressure tests are often eliminated the first
- Customers pay for new features in the first place, and not for tests
- But customers won’t accept an buggy system either
- Writing tests is not really a “fun task”
- Test are a sustainable, long term commitment, like an insurance

Write Tests to Enable Evolution

- This pattern basically is the rationale why to test at all
- Every change can potentially introduce a new defect or break the system
- Tests minimize those risks
- More important:
  - Automated, repeatable, persistent, documented -> well designed tests
  - Run tests after every change to verify its correctness
  - Use a mature testing framework (The “main-method” is not a mature testing framework)



Finished after 34,898 seconds

Runs: 13009/13009 Errors: 0 Failures: 0

- junit.framework.TestSuite
  - junit.framework.TestSuite
    - TestBagUtils
    - org.apache.commons.collections.TestClos
    - org.apache.commons.collections.TestColle
    - TestBufferUtils
    - TestEnumerationUtils
    - org.apache.commons.collections.TestFact
    - TestListUtils
    - TestMapUtils
    - org.apache.commons.collections.TestPred
    - TestSetUtils
    - org.apache.commons.collections.TestTrar
    - TestArrayStack
    - TestBeanMap
    - org.apache.commons.collections.TestBina
    - TestBoundedFifoBuffer
    - TestBoundedFifoBuffer2
    - TestCursorableLinkedList
    - TestDoubleOrderedMap
    - org.apache.commons.collections.TestExte
    - TestFastArrayList
    - TestFastArrayList1
    - TestFastHashMap
    - TestFastHashMap1
    - TestFastTreeMap
    - TestFastTreeMap1

```
public boolean addAll(int index, Collection c) {
    if(c.isEmpty()) {
        return false;
    } else if( size == index ||  size == 0) {
        return addAll(c);
    } else {
        Listable succ = getListableAt(index);
        Listable pred = (null == succ) ? null : succ.prev();
        Iterator it = c.iterator();
        while(it.hasNext()) {
            pred = insertListable(pred,succ,it.next());
        }
        return true;
    }
}
```

TestAllPackages (31.10.2006 15:04:14)

Element	Coverage	Covered Lines	Total Lines
java - commons-collections	79,5 %	10927	13738
org.apache.commons.collections	74,1 %	3842	5183
ArrayStack.java	86,5 %	32	37
BagUtils.java	86,7 %	13	15
BeanMap.java	72,4 %	155	214
BinaryHeap.java	87,6 %	127	145
BoundedFifoBuffer.java	93,2 %	82	88
BufferOverflowException.java	55,6 %	5	9
BufferUnderflowException.java	88,9 %	8	9
BufferUtils.java	30,8 %	4	13
ClosureUtils.java	93,9 %	31	33
CollectionUtils.java	92,4 %	293	317
ComparatorUtils.java	8,6 %	3	35
CursorableLinkedList.java	85,4 %	444	520

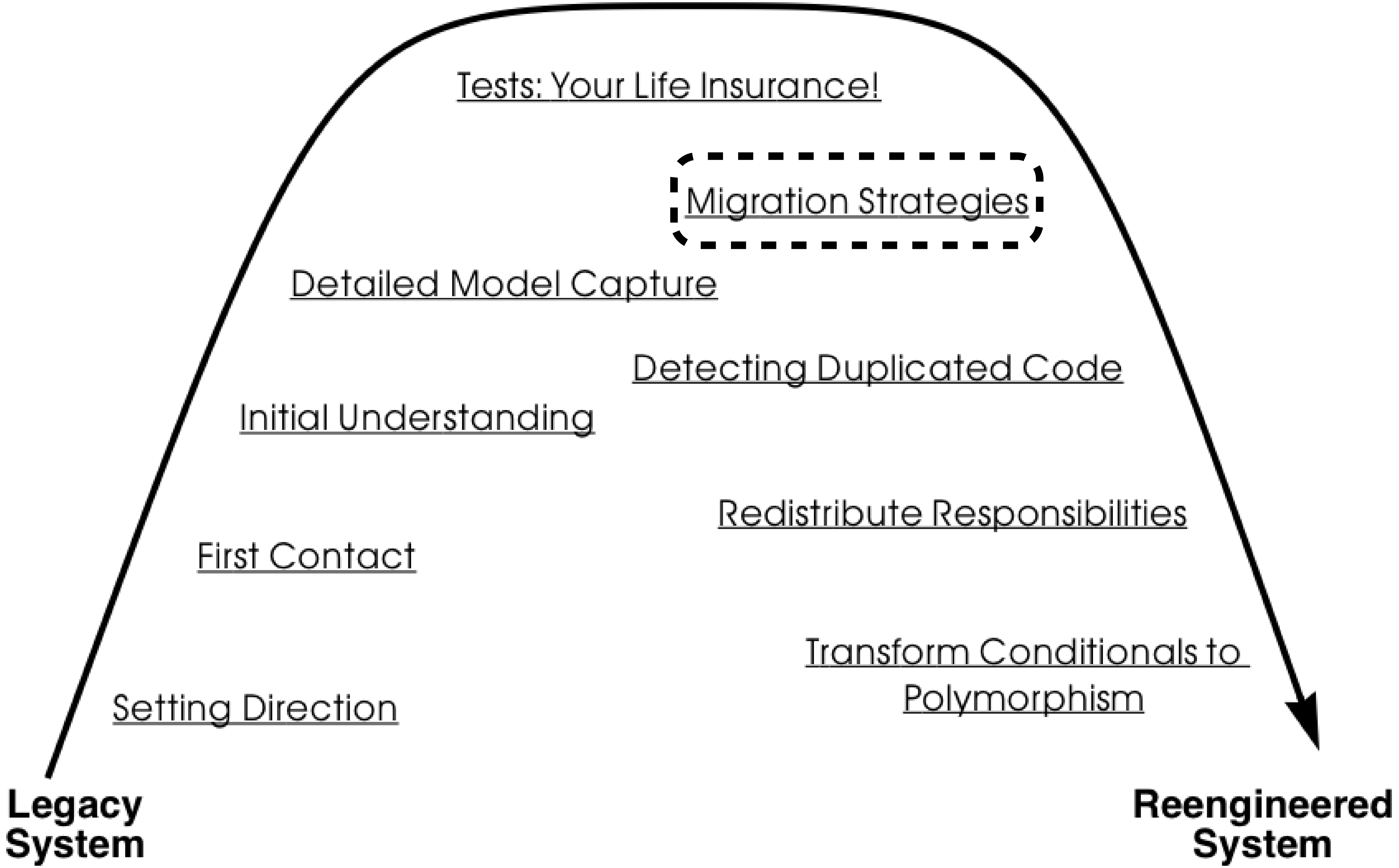
Eclipse and third party tools provide an excellent framework

Record Business Rules as Tests

- Business rules are important but often hidden and implicit in the code base
- It is not obvious which module is responsible for a given business rule
- Write test cases that encapsulate individual business rules
- Rules become explicit, therefore, the risk of loosing implicit knowledge because of developer turnover is reduced
- Again enables evolution: Whenever something is changed, we can check if the rules are still fulfilled correctly by simply running the tests
- Be aware: There might be a lot of different rules

Test the Interface, not the Implementation

- Test the external behavior, not the implementation
- Implementation details change often, interfaces are (more) stable
- Interface tests will survive changes to the implementation
- Focusing on the external behavior
- We don’t waste time in developing tests each time we make small changes to the system
- Black-Box testing



Migration Strategies (Ch. 7)

From new to old ... but how?

Migration Strategies (Ch.7)

- Migration to the new system happens while the old system is still running
- People are still using the old system and are skeptic about the new system
- Expects changes even while reengineering and deploying the new system
- Avoid a big Waterfall Project
- Migration of legacy system is an entire topic on its own

Migrate Incrementally

- *Step-wise* migration is the key
- *Avoid* the complexity and risks of *big-bang reengineering*
- Decompose the reengineering effort into parts; *deploy* those individual parts of the new system *gradually*
- Get *early feedback* from users
- Users *learn* the system *gradually*; they are not faced with one big change overnight
- You can *prioritize migration steps*: Deploy important parts first (and possibly re-iterate)

Always have a Running Version

- A running version is required:
  - For running tests
  - Gradually release the new system to the users
- A running version after the integration of changes builds confidence
- it is hard to get excited about the new system if it is not yet running
- If we break the system, we can always fall back to the last running version
- Continuously integrating changes is time consuming (use build and configuration management systems)
- The architecture must support a step-wise integration of of changes

Involve the Users

Maximize the acceptance of changes

Make a Bridge to the new Town

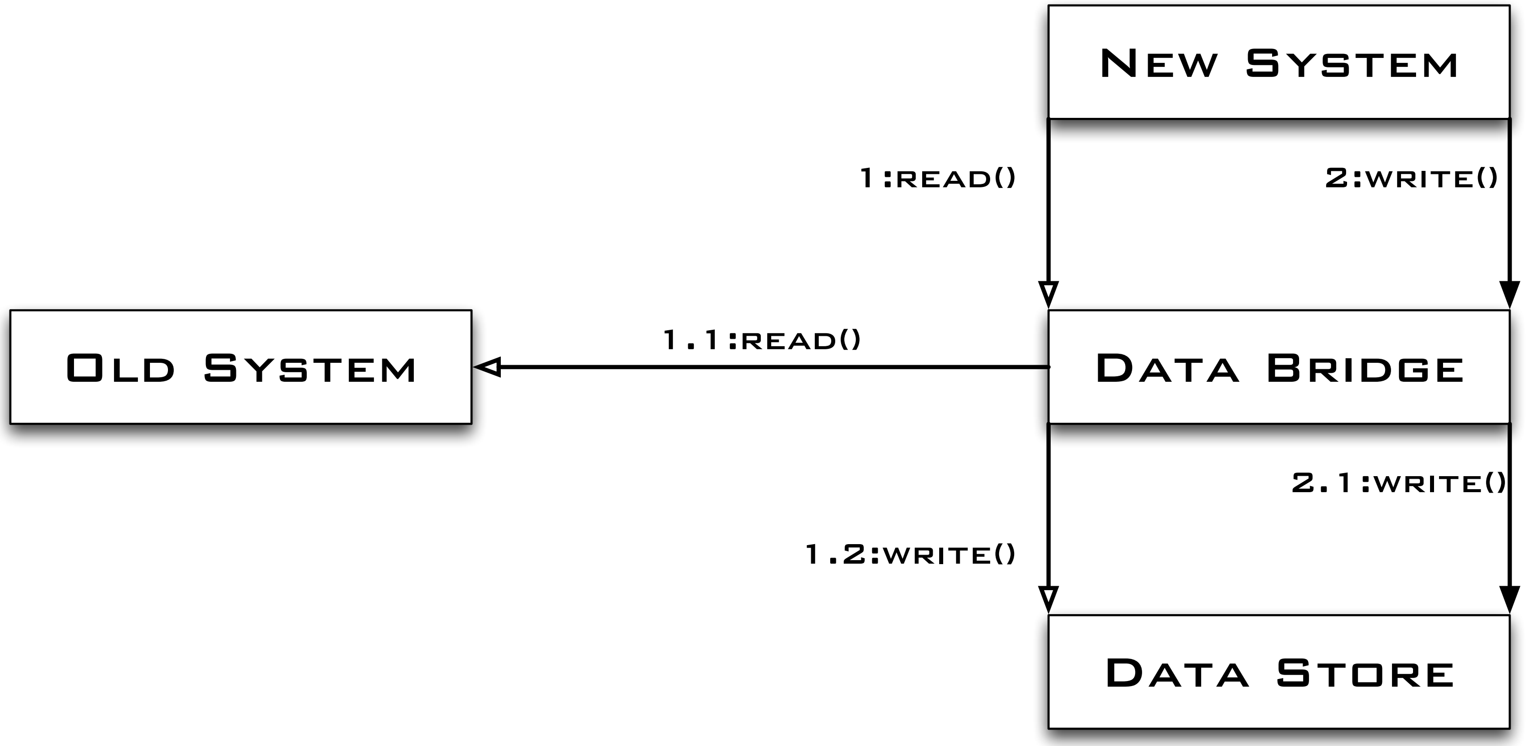
How to migrate the data?

Migrating Data is Difficult

- The old and new system are running in parallel
- Ensure that the data is transferred
- Ensure that that nothing is lost
- Ensure that the data remains uncorrupted
- Implements a “data bridge” that acts like a proxy for data manipulation

Data Bridge

- Data bridge redirects READ requests from the new system to the legacy database
- Data bridge makes data conversion
- Eventually data bridge automatically migrates requested data to the new system
- If necessary the old system code READS/WRITES to the new data store via bridge

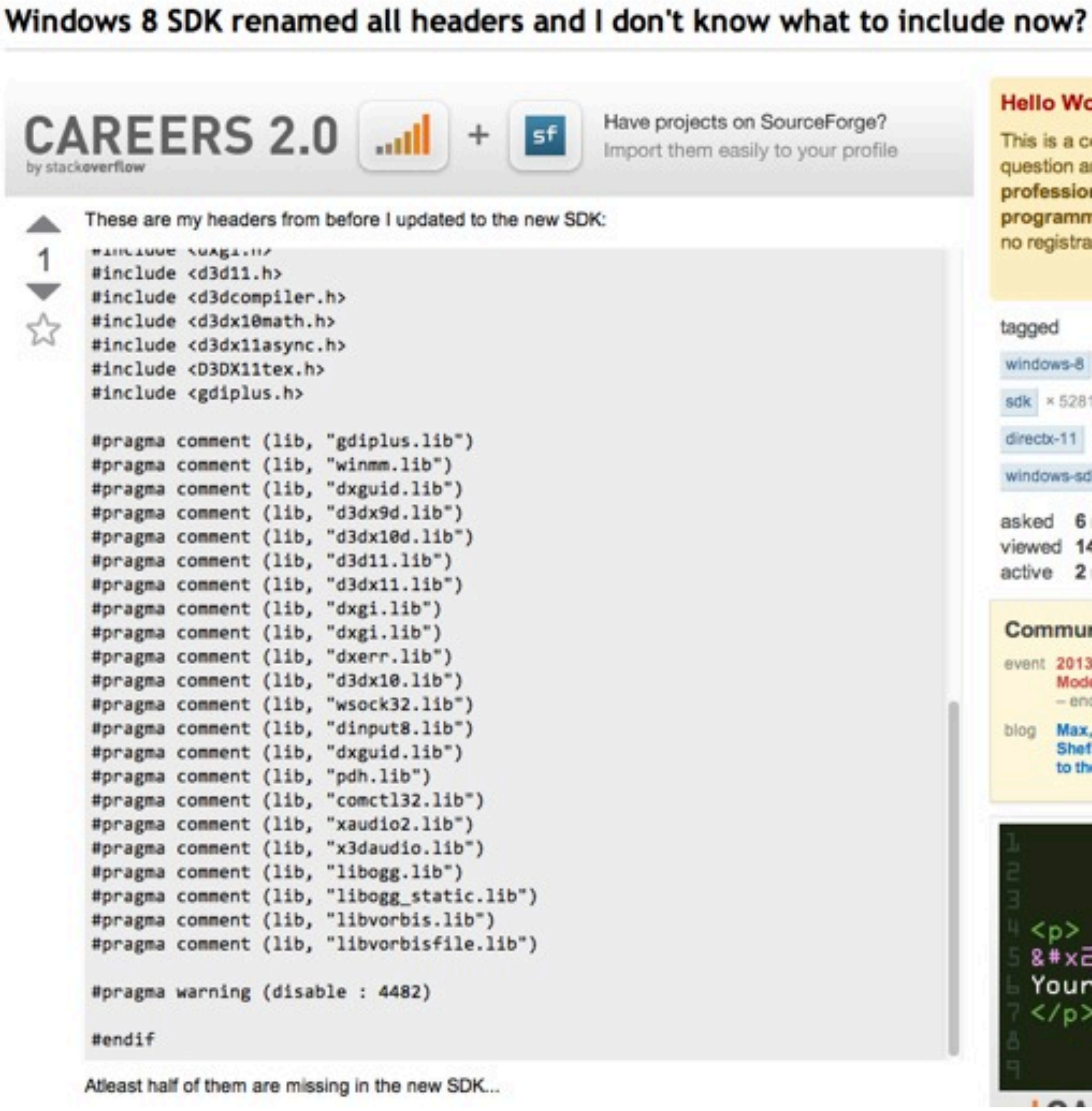


Deprecate obsolete Interfaces

- How to introduce new interfaces without invalidating all the clients?
- Avoid radical changes

Win 8 SDK and DirectX

- DirectX SDK part of Win 8 SDK
- Header files were renamed
- Math functions of the D3DX utility library are replaced
- DirectXMath should be used now
- Utility library for textures is completely replaced by 2 new frameworks DirectXTK and DirectXTex
- Code does not compile anymore
- You could still mix old and new



Windows 8 SDK renamed all headers and I don't know what to include now?

These are my headers from before I updated to the new SDK:

```
#include <dxgi.h>
#include <d3d11.h>
#include <d3dcompiler.h>
#include <d3dx10math.h>
#include <d3dx11async.h>
#include <D3DX11tex.h>
#include <gdiplus.h>

#pragma comment (lib, "gdiplus.lib")
#pragma comment (lib, "winmm.lib")
#pragma comment (lib, "dxguid.lib")
#pragma comment (lib, "d3dx9d.lib")
#pragma comment (lib, "d3dx10d.lib")
#pragma comment (lib, "d3d11.lib")
#pragma comment (lib, "d3dx11.lib")
#pragma comment (lib, "dxgi.lib")
#pragma comment (lib, "dxgi.lib")
#pragma comment (lib, "dxerr.lib")
#pragma comment (lib, "d3dx10.lib")
#pragma comment (lib, "wsock32.lib")
#pragma comment (lib, "dinput8.lib")
#pragma comment (lib, "dxguid.lib")
#pragma comment (lib, "pdh.lib")
#pragma comment (lib, "comctl32.lib")
#pragma comment (lib, "xaudio2.lib")
#pragma comment (lib, "x3daudio.lib")
#pragma comment (lib, "libogg.lib")
#pragma comment (lib, "libogg_static.lib")
#pragma comment (lib, "libvorbis.lib")
#pragma comment (lib, "libvorbisfile.lib")

#pragma warning (disable : 4482)

#endif
```

Atleast half of them are missing in the new SDK...

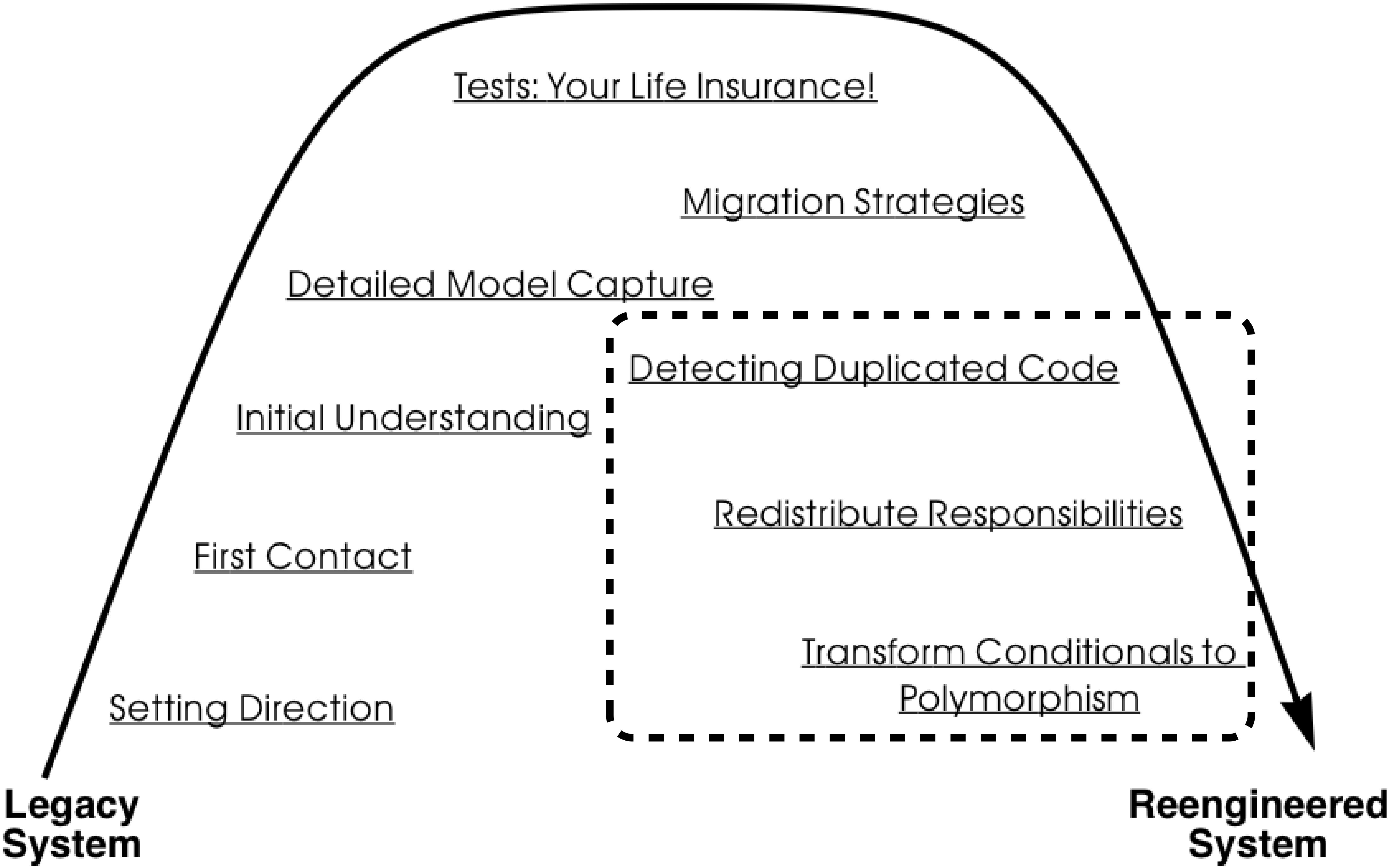
Deprecate Obsolete Interfaces

- How to introduce new interfaces without invalidating all the clients?
- Leaving old interfaces in code will blow up the API
- Makes maintenance difficult
- Clients will most likely stick with the old interfaces
- Describe old interfaces as obsolete
- Give clients some time to react

Java

- Annotation *@deprecated,* part of Java Doc
- Labels classes and methods that are no longer supported
- Code still compiles and runs
- Compiler issues a warning
- You set a link to redirect clients to the new method/class

```
/**
 * @deprecated  As of release 1.3, replaced by
{@link #getPreferredSize()}
 */
@Deprecated public Dimension preferredSize() {
return getPreferredSize();
}
```



Legacy System

Setting Direction

First Contact

Initial Understanding

Detailed Model Capture

Tests: Your Life Insurance!

Migration Strategies

Detecting Duplicated Code

Redistribute Responsibilities

Transform Conditionals to Polymorphism

Reengineered System

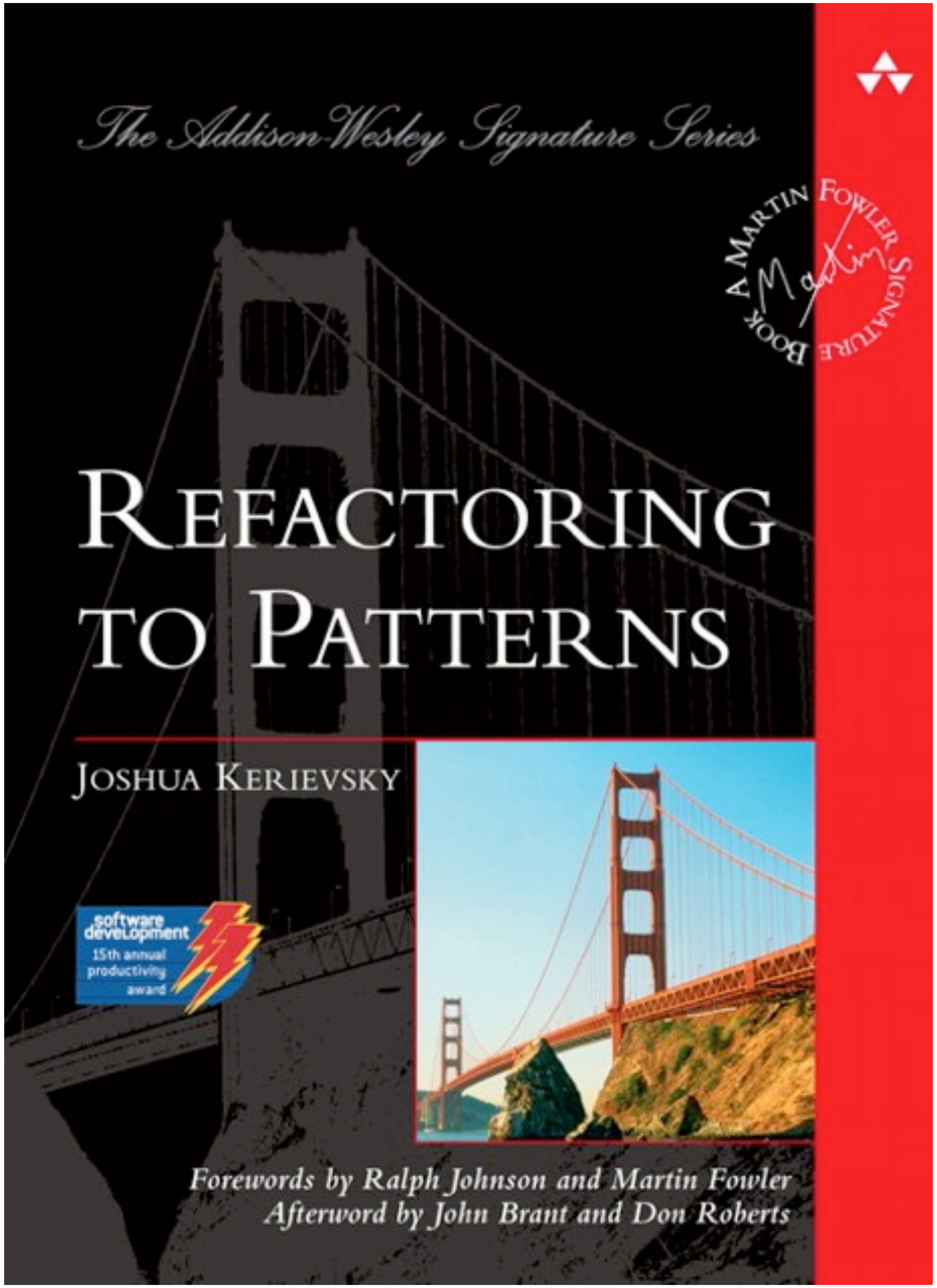
Refactoring

Refactoring

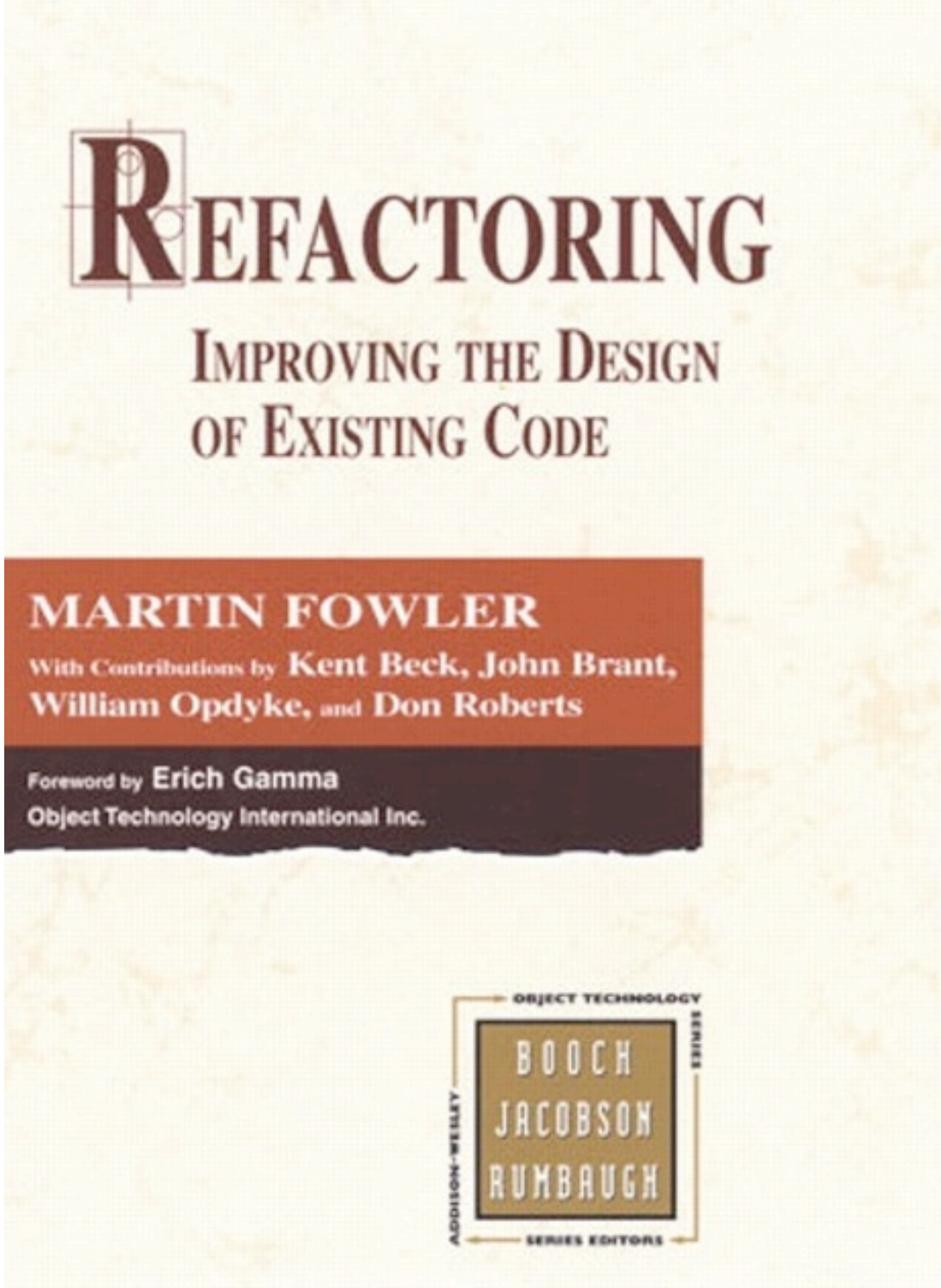
Make your code look nice

Refactoring

- Process of *improving* the *internal structure* of the code
- During this process the *external behavior*, i.e., the *functionality*, of the system DOES NOT *change*
- Part of the reengineering cycle
- Refactoring can also happen is a smaller context (daily work) outside of a big reengineering project
- Notice: At the end of the entire reengineering project the system may implement new features



Refactoring to Patterns *by*
Joshua Kerievsky
Addison-Wesley Longman , 2044



Refactoring: Improving the Design of existing Code *by*
Martin Fowler
Addision-Wesley Professional , 1999

Refactoring Workflow

1. Make sure your tests pass (You need to ensure that the code behaves the same after refactoring)

2. Find the bad code

3. Find a solution how to make it look nice

4. Modify the code

5.Run unit tests to verify the correctness of the refactoring

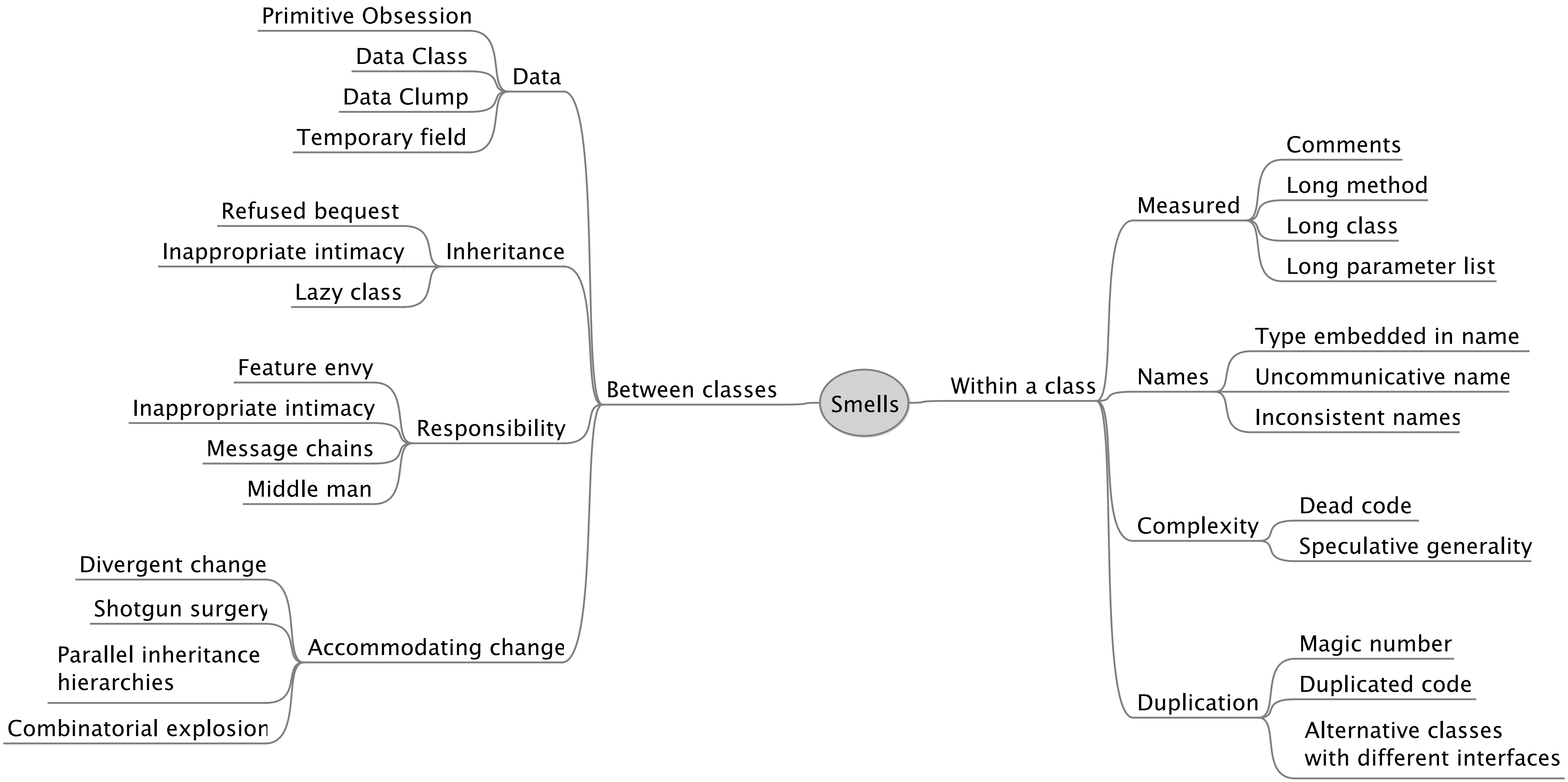
6. Repeat step 1-5 until all bad code is eliminated

What is exactly is *bad*?

- Indications to start a reengineering project:
    - Lot of bug fixing, long time to market cycles, evolution of a system gets out of control, ...
- But how can we identify *bad code*?
- Bad code can turn even small changes into a difficult, large nightmare
- *Bad smells* are indications for refactoring potential on source code level
- In practice bad smells are violations of good design principles & design heuristics

Bad Smell Detection

- Difficult because smells can affect more than just one class:
  - Sub-optimal inheritance structure
  - Dependency Inversion Principle (high-level module should not depend on low-level modules)
- There is tool support to find bad smells
- Detection is often based on metrics
- Lecture on *Software Visualization*



- Smells
  - Between classes
    - Data
      - Primitive Obsession
      - Data Class
      - Data Clump
      - Temporary field
    - Inheritance
      - Refused bequest
      - Inappropriate intimacy
      - Lazy class
    - Responsibility
      - Feature envy
      - Inappropriate intimacy
      - Message chains
      - Middle man
    - Accommodating change
      - Divergent change
      - Shotgun surgery
      - Parallel inheritance hierarchies
      - Combinatorial explosion
  - Within a class
    - Measured
      - Comments
      - Long method
      - Long class
      - Long parameter list
    - Names
      - Type embedded in name
      - Uncommunicative name
      - Inconsistent names
    - Complexity
      - Dead code
      - Speculative generality
    - Duplication
      - Magic number
      - Duplicated code
      - Alternative classes with different interfaces

Bad Smell Overview

Fowler (1999)

Refactorings

- Moving features between objects
  - Move method
  - Move field
  - Extract class
  - Inline class
  - Hide delegate
  - Remove middle man
  - Introduce foreign method
  - Introduce local extension
- Simplifying conditional expressions
  - Decompose conditional
  - Consolidate conditional expression
  - Consolidate duplicate conditional fragments
  - Remove control flag
  - replace nested conditional with guard clauses
  - Replace conditional with polymorphism
  - Introduce null object
  - Introduce assertion
- Dealing with generalization
  - Pull up field
  - Pull up method
  - Pull up contructor body
  - Extract subclass
  - Extract superclass
  - Extract interface
  - Collapse hierarchy
  - Form template method
  - Replace inheritance with delegation
  - Replace delegation with inheritance
- Composing Methods
  - Inline method
  - Inline temp
  - Replace temp with query
  - Introduce explaining variable
  - Split temporary variable
  - Remove assignment to parameters
  - Replace method with method object
  - Substitute algorithm
- Organizing data
  - Self encapsulate field
  - Replace data value with object
  - Change value to reference
  - Change reference to value
  - Replace array with object
  - Duplicate observed data
  - Change unidirectional association to bidirectiona
  - Change bidirectional association to unidirectiona
  - Replace magic number with symbolic constant
  - Encapsulate field
  - Replace record with data class
  - Replace type code with class
  - Replace type code with subclasses
  - Replace type code with state/strategy
  - Replace subclass with fields
- Making method calls simpler
  - Rename method
  - Add parameter
  - Remove parameter
  - Separate query from modifier
  - Parameterize method
  - Replace parameter with explicit method
  - Preserve whole object
  - Replace parameter with method
  - Replace error code with exception
  - Replace exception with test

Refactoring's

Fowler (199)

When to Refactor?

- When you add functionality
- When you learn something about the code
- When you fix a bug
- Code Smells
- You (should) do it all the time

Refactoring Examples (with Ecplise)

Extract Method

- Gather a block of code statements and move it in a new method
- Improves readability of the code
- Summarizes the intent of code in a single meaningful method name
- Reduce the length of a method (Bad Smell: *Long method*)
- Remove duplicated code (improves code reuse)
- Smaller method are generally easier to maintain than really big ones
- Rule of Thumb: Only extract a new method if you can find a good name
- Opposite: Inline method

Refactorings

- Moving features between objects
  - Move method
  - Move field
  - Extract class
  - Inline class
  - Hide delegate
  - Remove middle man
  - Introduce foreign method
  - Introduce local extension
- Simplifying conditional expressions
  - Decompose conditional
  - Consolidate conditional expression
  - Consolidate duplicate conditional fragments
  - Remove control flag
  - replace nested conditional with guard clauses
  - Replace conditional with polymorphism
  - Introduce null object
  - Introduce assertion
- Dealing with generalization
  - Pull up field
  - Pull up method
  - Pull up contructor body
  - Extract subclass
  - Extract superclass
  - Extract interface
  - Collapse hierarchy
  - Form template method
  - Replace inheritance with delegation
  - Replace delegation with inheritance
- Composing Method
  - Inline method
  - Inline temp
  - Replace temp with query
  - Introduce explaining variable
  - Split temporary variable
  - Remove assignment to parameters
  - Replace method with method object
  - Substitute algorithm
- Organizing data
  - Self encapsulate field
  - Replace data value with object
  - Change value to reference
  - Change reference to value
  - Replace array with object
  - Duplicate observed data
  - Change unidirectional association to bidirectiona
  - Change bidirectional association to unidirectiona
  - Replace magic number with symbolic constant
  - Encapsulate field
  - Replace record with data class
  - Replace type code with class
  - Replace type code with subclasses
  - Replace type code with state/strategy
  - Replace subclass with fields
- Making method calls simpler
  - Rename method
  - Add parameter
  - Remove parameter
  - Separate query from modifier
  - Parameterize method
  - Replace parameter with explicit method
  - Preserve whole object
  - Replace parameter with method
  - Replace error code with exception
  - Replace exception with test

Refactoring's

Fowler (199)

Move Method

- Where to put functionality my design?
- Fundamental aspect of object-oriented design
- *Class Responsibility Card* by Cunningham and Beck[1]
- Keep behavior and data together
- Slim down the interface of a class

[1] A laboratory for teaching object oriented thinking *by*
Kent Beck and Cunningham Ward
@ OOPSLA Conference, 1989

Student	
Responsibilities	*Collaborators*
Represents a student of the university. Holds all the necessary data of an individual student Encapsulates the data but provides access to data via interface methods	

Student Admin	
Responsibilities	*Collaborators*
Provides back-end functionality of the student administration	SAP Database interface for read and write access to student data

- **Refactorings**
  - Moving features between objects
    - Move method
    - Move field
    - Extract class
    - Inline class
    - Hide delegate
    - Remove middle man
    - Introduce foreign method
    - Introduce local extension
  - Simplifying conditional expressions
    - Decompose conditional
    - Consolidate conditional expression
    - Consolidate duplicate conditional fragments
    - Remove control flag
    - replace nested conditional with guard clauses
    - Replace conditional with polymorphism
    - Introduce null object
    - Introduce assertion
  - Dealing with generalization
    - Pull up field
    - Pull up method
    - Pull up contructor body
    - Extract subclass
    - Extract superclass
    - Extract interface
    - Collapse hierarchy
    - Form template method
    - Replace inheritance with delegation
    - Replace delegation with inheritance
  - Composing Methods
    - Inline method
    - Inline temp
    - Replace temp with query
    - Introduce explaining variable
    - Split temporary variable
    - Remove assignment to parameters
    - Replace method with method object
    - Substitute algorithm
  - Organizing data
    - Self encapsulate field
    - Replace data value with object
    - Change value to reference
    - Change reference to value
    - Replace array with object
    - Duplicate observed data
    - Change unidirectional association to bidirectiona
    - Change bidirectional association to unidirectiona
    - Replace magic number with symbolic constant
    - Encapsulate field
    - Replace record with data class
    - Replace type code with class
    - Replace type code with subclasses
    - Replace type code with state/strategy
    - Replace subclass with fields
  - Making method calls simpler
    - Rename method
    - Add parameter
    - Remove parameter
    - Separate query from modifier
    - Parameterize method
    - Replace parameter with explicit method
    - Preserve whole object
    - Replace parameter with method
    - Replace error code with exception
    - Replace exception with test

Refactoring's

Fowler (199)

Organize Data

- Data inherently involves (low level) implementation details
- Datatypes, data structures
- Needs to satisfy constraints
- It often accessed/modified by many functions
- Hide implementation details and provide a unified access to data
- Last Lecture: Magic Numbers
- Other examples: Replace Type Code with Class, Encapsulate field, ...

- Refactorings
  - Moving features between objects
    - Move method
    - Move field
    - Extract class
    - Inline class
    - Hide delegate
    - Remove middle man
    - Introduce foreign method
    - Introduce local extension
  - Simplifying conditional expressions
    - Decompose conditional
    - Consolidate conditional expression
    - Consolidate duplicate conditional fragments
    - Remove control flag
    - replace nested conditional with guard clauses
    - Replace conditional with polymorphism
    - Introduce null object
    - Introduce assertion
  - Dealing with generalization
    - Pull up field
    - Pull up method
    - Pull up contructor body
    - Extract subclass
    - Extract superclass
    - Extract interface
    - Collapse hierarchy
    - Form template method
    - Replace inheritance with delegation
    - Replace delegation with inheritance
  - Composing Methods
    - Inline method
    - Inline temp
    - Replace temp with query
    - Introduce explaining variable
    - Split temporary variable
    - Remove assignment to parameters
    - Replace method with method object
    - Substitute algorithm
  - Organizing data
    - Self encapsulate field
    - Replace data value with object
    - Change value to reference
    - Change reference to value
    - Replace array with object
    - Duplicate observed data
    - Change unidirectional association to bidirectiona
    - Change bidirectional association to unidirectiona
    - Replace magic number with symbolic constant
    - Encapsulate field
    - Replace record with data class
    - Replace type code with class
    - Replace type code with subclasses
    - Replace type code with state/strategy
    - Replace subclass with fields
  - Making method calls simpler
    - Rename method
    - Add parameter
    - Remove parameter
    - Separate query from modifier
    - Parameterize method
    - Replace parameter with explicit method
    - Preserve whole object
    - Replace parameter with method
    - Replace error code with exception
    - Replace exception with test

Refactoring's

Fowler (199)

Introduce Parameter Object

- Method signatures with many parameters are difficult to read
- In many cases several parameters carry a certain data semantic
- Create a class to group all parameters into a single object
- Purpose of parameter object is to pass values into the method
- Easier to add new values ->add another field to parameter object

Real World Example: DirectX API

```
D3D11_BUFFER_DESC vertexBufferDesc; //Parameter object


//Set values of parameter object

vertexBufferDesc.Usage = D3D11_USAGE_DEFAULT; //Example of Replace Type Code

vertexBufferDesc.ByteWidth = sizeof(VertexType) * m_vertexCount;

vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;

vertexBufferDesc.CPUAccessFlags = 0;

vertexBufferDesc.MiscFlags = 0;

vertexBufferDesc.StructureByteStride = 0;


//Pass parameter object to method

Direct3D11device->CreateBuffer(&vertexBufferDesc,....);
```

D3D11_USAGE_DEFAULT

```
//reflects whether a resource is accessible by the CPU and/or the graphics processing unit (GPU).

typedef enum D3D11_USAGE {

  D3D11_USAGE_DEFAULT    = 0,

  D3D11_USAGE_IMMUTABLE  = 1,

  D3D11_USAGE_DYNAMIC    = 2,

  D3D11_USAGE_STAGING    = 3

} D3D11_USAGE;
```

- Refactorings
  - Moving features between objects
    - Move method
    - Move field
    - Extract class
    - Inline class
    - Hide delegate
    - Remove middle man
    - Introduce foreign method
    - Introduce local extension
  - Simplifying conditional expressions
    - Decompose conditional
    - Consolidate conditional expression
    - Consolidate duplicate conditional fragments
    - Remove control flag
    - replace nested conditional with guard clauses
    - Replace conditional with polymorphism
    - Introduce null object
    - Introduce assertion
  - Dealing with generalization
    - Pull up field
    - Pull up method
    - Pull up contructor body
    - Extract subclass
    - Extract superclass
    - Extract interface
    - Collapse hierarchy
    - Form template method
    - Replace inheritance with delegation
    - Replace delegation with inheritance
  - Composing Methods
    - Inline method
    - Inline temp
    - Replace temp with query
    - Introduce explaining variable
    - Split temporary variable
    - Remove assignment to parameters
    - Replace method with method object
    - Substitute algorithm
  - Organizing data
    - Self encapsulate field
    - Replace data value with object
    - Change value to reference
    - Change reference to value
    - Replace array with object
    - Duplicate observed data
    - Change unidirectional association to bidirectiona
    - Change bidirectional association to unidirectiona
    - Replace magic number with symbolic constant
    - Encapsulate field
    - Replace record with data class
    - Replace type code with class
    - Replace type code with subclasses
    - Replace type code with state/strategy
    - Replace subclass with fields
  - Making method calls simpler
    - Rename method
    - Add parameter
    - Remove parameter
    - Separate query from modifier
    - Parameterize method
    - Replace parameter with explicit method
    - Preserve whole object
    - Replace parameter with method
    - Replace error code with exception
    - Replace exception with test

Refactoring's

Fowler (199)

Making Conditional Expressions Easier

- Application logic can be complex and difficult to get right
- Logic is central and changes often
- In OO conditional behavior is handled by polymorphism
- Logic is encapsulated in objects
- Less complex conditional statements and more flexibility
- Logic is decentralized across different classes: Runtime vs. Static

Problems of Refactoring

- Taken to far:
    - Risk of over-engineering, “desperately” searching for refactoring opportunities
- Don’t refactor if there are not any running tests
- Databases are difficult to refactor
- Refactoring changes API
    - Choose appropriate migration strategy
    - Keep old interface, but flag as deprecated

Refactoring Reading Material

- http://sourcemaking.com/refactoring