Software Reengineering Working with Legacy Code

Martin Pinzger – Andy Zaidman
Delft University of Technology

TU Delft

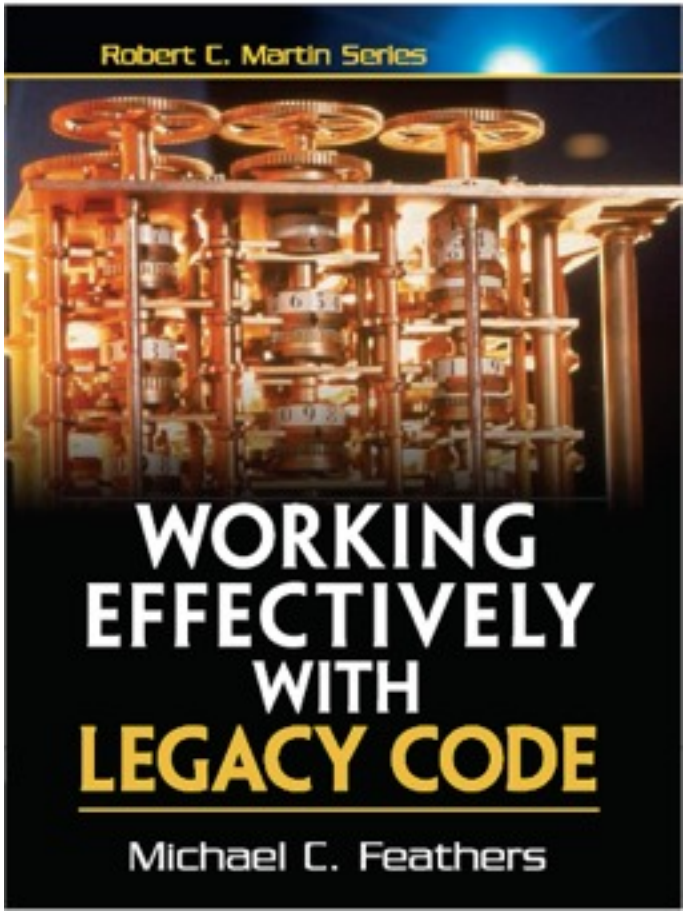
SERG

Outline

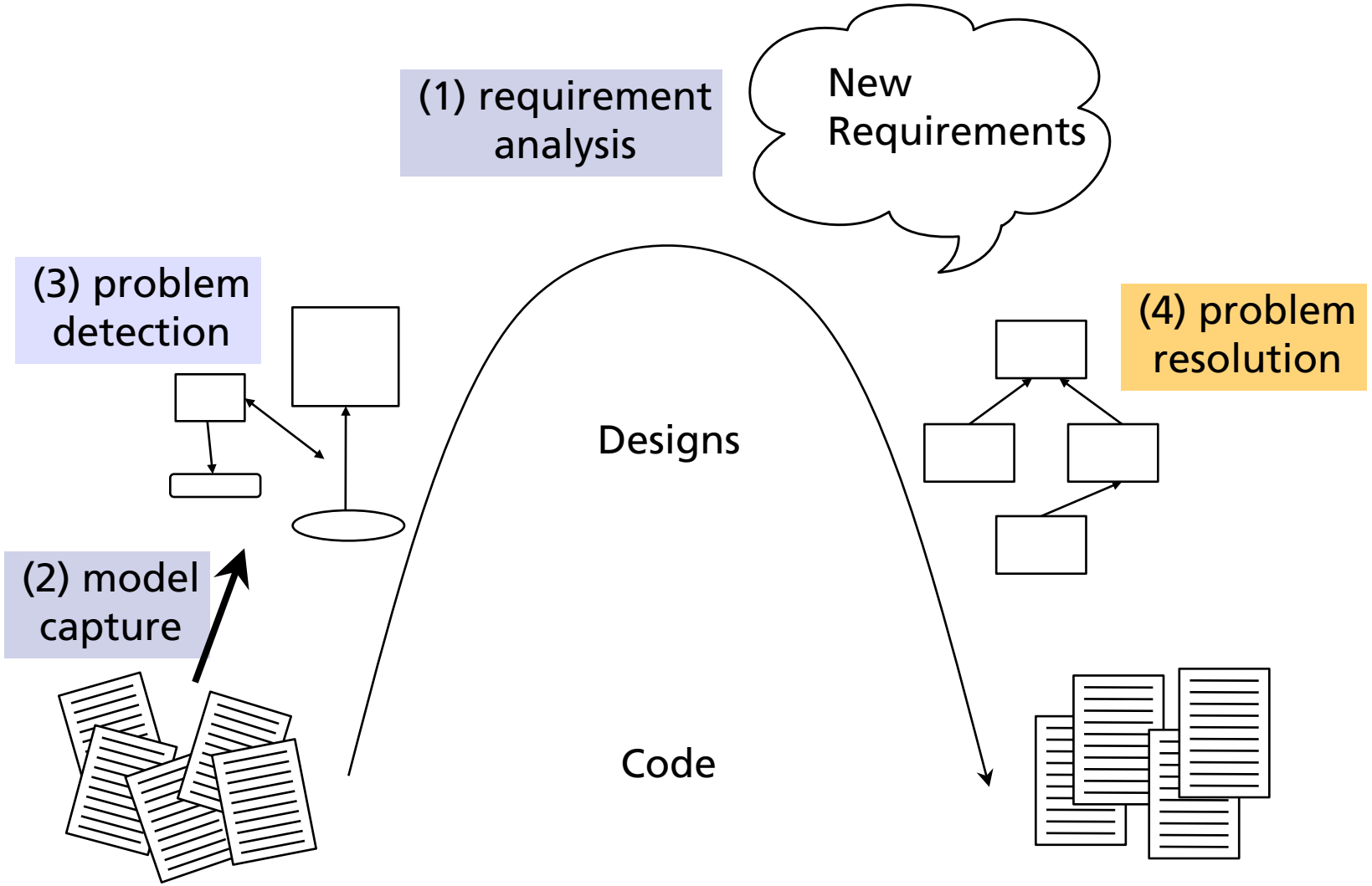
Introduction

Breaking dependencies

Strategies for breaking dependencies



The Reengineering Life-Cycle



The Legacy Code Dilemma

"When we change code, we should have tests in place. To put tests in place, we often have to change code."

[Feathers 2005]

Most applications are glued together

Singletons (a.k.a. global variables)

If there can only be one of an object you'd better hope that it is good for your tests too

Internal instantiation

When a class creates an instance of a hard coded class, you'd better hope that class runs well in tests

Concrete dependency

When a class uses a concrete class, you'd better hope that class lets you know what is happening to it

How to get my tests in?

"By breaking dependencies"

Sensing

When we can't access values our code computes

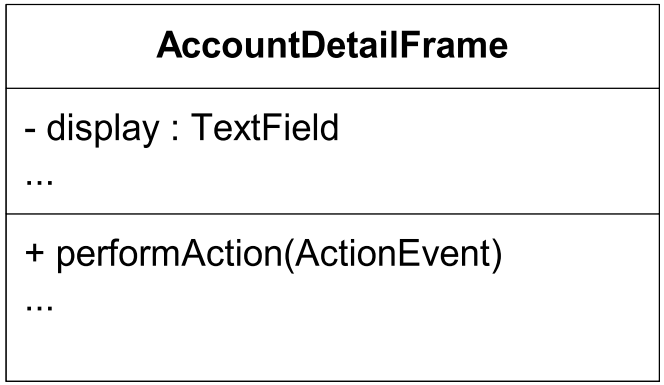
Separation

When we can't even get a piece of code into a test harness to run

The legacy code change algorithm

1. Identify change points
2. Find test points
3. Break dependencies
4. Write tests
5. Make changes and refactor

Example: Breaking dependencies



An ugly implementation

```
public class AccountDetailFrame extends Frame implements ActionListener,
    WindowListener {
  private TextField display = new TextField(10);
  …
  private AccountDetailFrame(…) { … }

  public void performAction(ActionEvent event) {
    String source = (String)event.getActionCommand();
    if (source.equals("project activity")) {
      DetailFrame detailDisplay = new DetailFrame();
      detailDisplay.setDescription(getDetailText() + " " + getProjectText());
      detailDisplay.show();
      String accountDescription = detailDisplay.getAccountSymbol();
      accountDescription += ": ";
      …
      display.setText(accountDescription);
      …
    }
  }
}
```

Where are the dependencies?

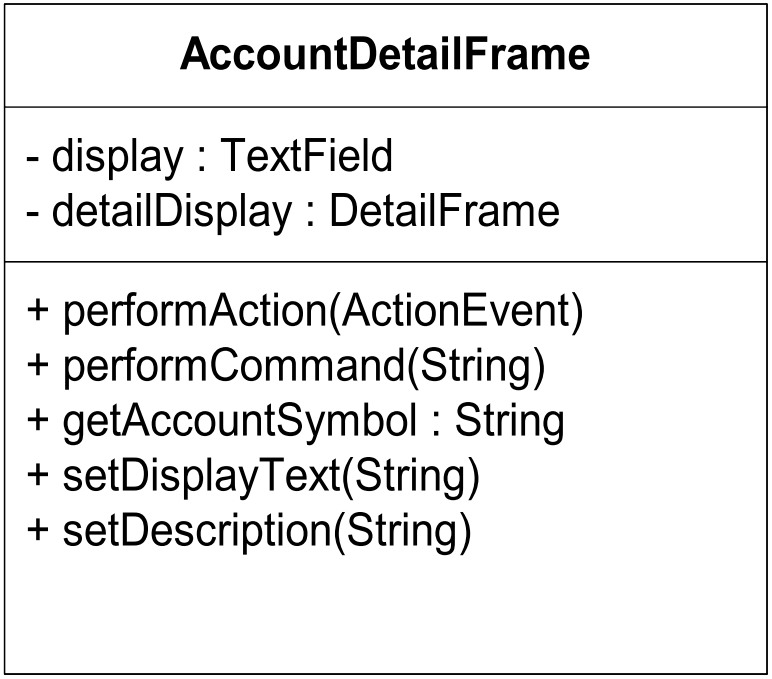
After extract method

```
public class AccountDetailFrame extends Frame implements ActionListener,
    WindowListener {
  private TextField display = new TextField(10);
  …
  private AccountDetailFrame(…) { … }
  public void performAction(ActionEvent event) {
    performCommand((String) event.getActionCommand());
  }

  void performCommand(String source) {
    if (source.equals("project activity")) {
      DetailFrame detailDisplay = new DetailFrame();
      detailDisplay.setDescription(getDetailText() + " " + getProjectText());
      detailDisplay.show();
      String accountDescription = detailDisplay.getAccountSymbol();
      accountDescription += ": ";
      …
      display.setText(accountDescription);
      …
    }
  }
}
```

After more refactoring

```
public class AccountDetailFrame extends Frame implements ActionListener,
    WindowListener {
  private TextField display = new TextField(10);
  private DetailFrame detailDisplay;
  …
  private AccountDetailFrame(…) { … }
  public void performAction(ActionEvent event) {
    performCommand((String)event.getActionCommand());
  }

  void performCommand(String source) {
    if (source.equals("project activity")) {
      setDescription(getDetailText() + "" + getProjectText());
      …
      String accountDescripton = getAccountSymbol();
      accountDescription += ": ";
      …
      setDisplayText(accountDescription);
      …
    }
  }
}
```

The Aftermath ...

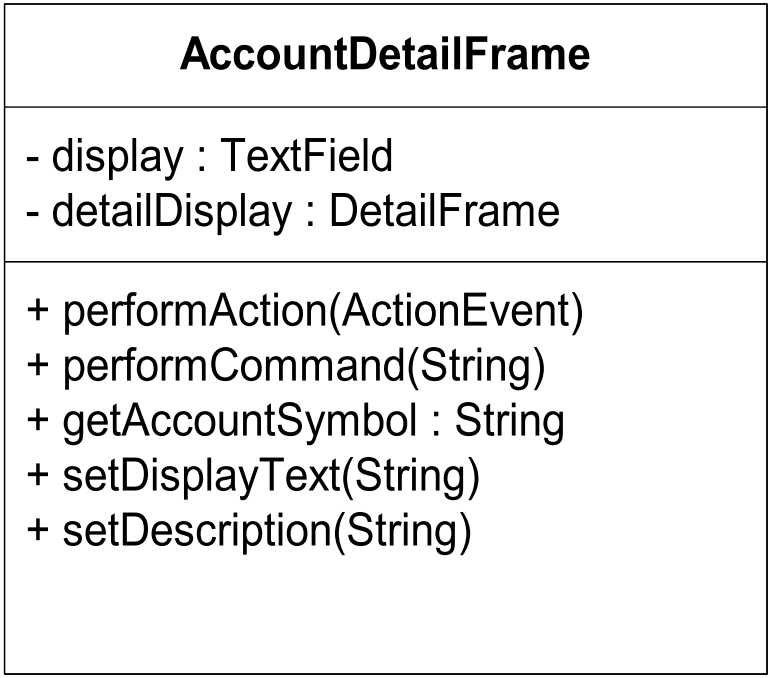


We can subclass AccountDetailFrame, and override getAccountSymbol, setDisplayText, and setDescription to sense our work

Adding a test

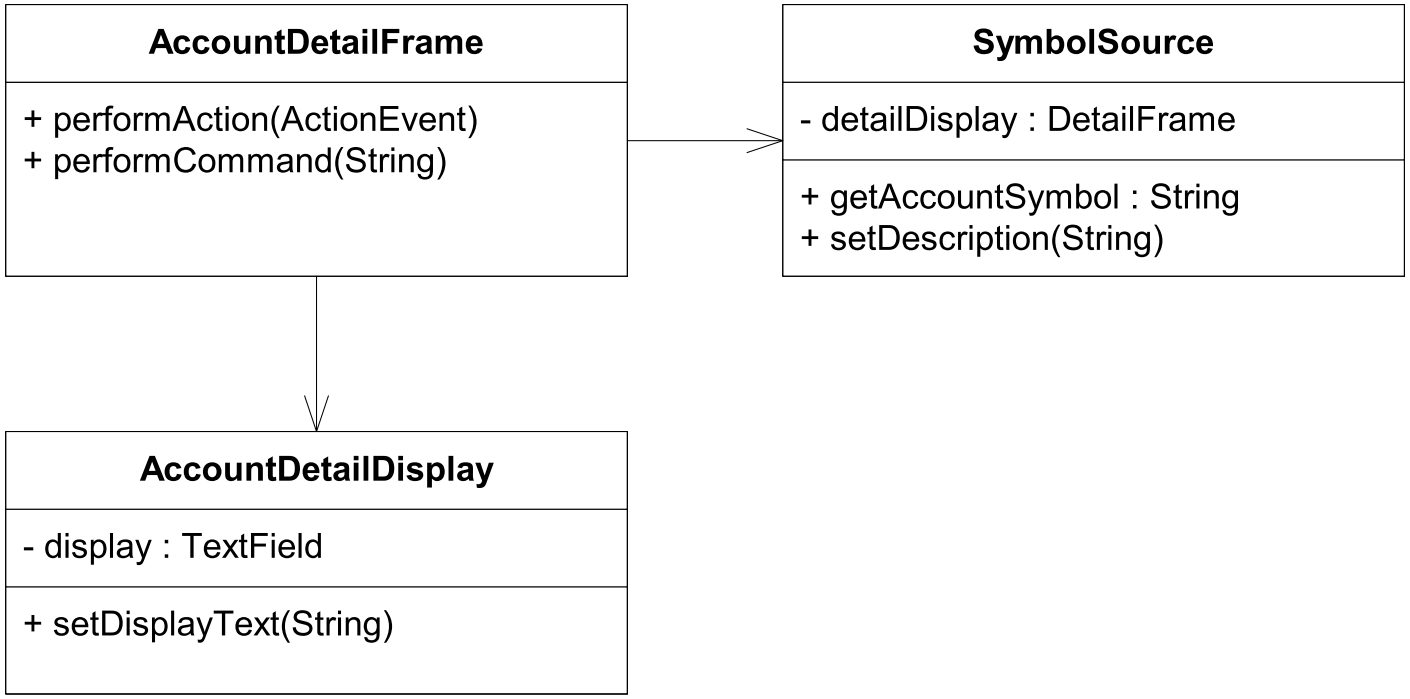
```
public TestingAccountDetailFrame extends AccountDetailFrame {
  ...
}

public class AccountDetailFrameTest {
  ...
  @Test
  public void testPerformCommand() {
    TestingAccountDetailFrame frame = new TestingAccountDetailFrame();

    frame.accountSymbol = “SYM”;
    frame.performCommand(“project activity”);
    assertEquals(“06 080 012”, frame.getDisplayText());
  }
}
```

More Aftermath ...



Making it better

We can make the design better by extracting classes for those separate responsibilities



Better vs. Best

“Best is the enemy of good” – Voltaire (paraphrased)

Strategies for breaking dependencies

- Lean on tools
  - Use tools with automated refactorings
  - Mock libraries
- If manual, then
  - Lean on the compiler
    - Use the compiler to report points of change by deliberately producing errors
  - Preserve signatures
    - Favor refactorings which allow you to cut/copy and paste signatures without modification
  - Single goal editing
    - Do one thing at a time!

I can’t get this class in a test harness

Common problems

Objects of the class can’t be created easily

The test harness won’t easily build with the class in it

The constructor we need to use has bad side effects

Significant work happens in the constructor, and we need to sense it

Irritating parameters

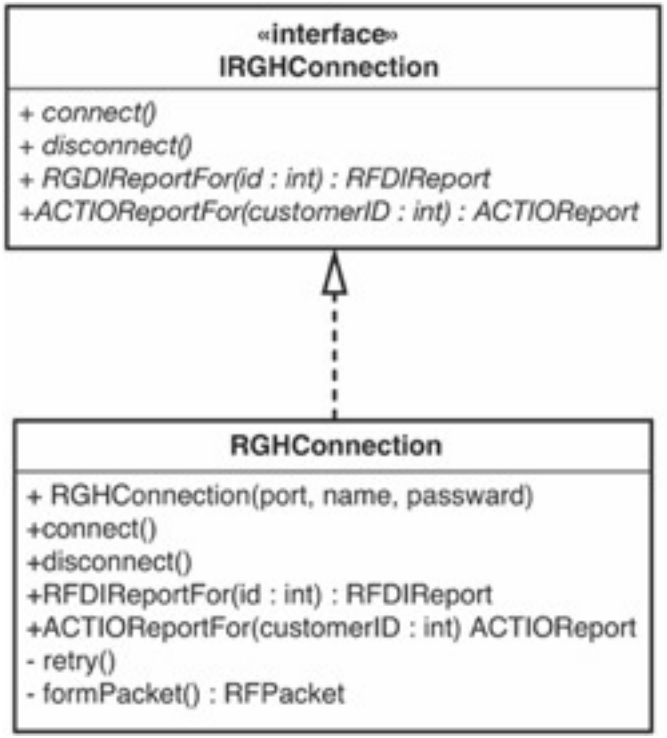
How am I going to construct these parameters for the test?

```
public class CreditValidator {

  public CreditValidator(RGHConnection connection,
    CreditMaster master,
    String validatorID) {
    ...
  }

  Certificate validateCustomer(Customer customer)
      throws InvalidCredit {
    ...
  }
  ...
}
```

Setting up network connection is not possible…

Irritating parameter: Solution 1

Extract interface + create FakeConnection class



+

```
public class FakeConnection
    implements IRGHConnection {
  public RFDIReport report;
  public void connect() {}
  public void disconnect() {}
  …
}
```

Irritating parameter: Solution 2

- Pass null
  - If an object requires a parameter that is hard to construct
    - If the parameter is used during your test execution an exception will be thrown
    - You can then still reconsider to construct a real object
- Variant solution “null object”
  - A sibling to the original class with no real functionality (returns default values).

Hidden dependency

Consider this C++ constructor:

```
mailing_list_dispatcher::mailing_list_dispatcher()
    : service(new mail_service), status(MAIL_OKAY)
{
  const int client_type = 12;
  service->connect();
  status = MAIL_OFFLINE;
  ...
}
```

This constructor relies on the class **mail_service** in the initialization list

We don't want to initialize the mail_service, because then we connect to the network and start sending actual mails…

Hidden dependency: Solution

Parameterize constructor

```
mailing_list_dispatcher::mailing_list_dispatcher
  (mail_service *service) : status(MAIL_OKAY)
```

Big improvement? (Yes)

- Allows for introducing a fake mail service
  - Extract interface for mail_service
  - Introduce fake class that senses the things we do

The construction blob

```
class WatercolorPane {
public:
  WatercolorPane(Form *border, WashBrush *brush,
  Pattern *backdrop) {
    ...
    anteriorPanel = new Panel(border);
    anteriorPanel->setBorderCol(brush->getForeColor());
    backgroundPanel = new Panel(border, backdrop);
    cursor = new FocusWidget(brush, backgroundPanel);
  }
}
```

How to sense the cursor?

The construction blob: Solution 1

Extract and Override Factory Method

If we have a refactoring tool

The construction blob: Solution 2

Supersede Instance Variable

```
// C++
void supersedeCursor(FocusWidget *newCursor) {
  delete cursor;
  cursor = newCursor;
}
```

After the object has been constructed, swap in another instance of cursor

- Be careful not to use superseding in production code

Using Extract Interface create a fake object for FocusWidget that you can use to sense

Irritating global dependency

```
public class Facility {
  private Permit basePermit;

  public Facility(int facilityCode, String owner, PermitNotice notice) throws
   PermitViolation {
    Permit associatedPermit =
      PermitRepository.getInstance().findAssociatedPermit(notice);
    if (associatedPermit.isValid() && !notice.isValid()) {
      basePermit = associatedPermit;
    } else if (!notice.isValid()) {
      Permit permit = new Permit(notice);
      permit.validate();
      basePermit = permit;
    } else {
      throw new PermitViolation(permit);
    }
  }
  ...
}
```

PermitRepository is a Singleton!

Where is the irritating global dependency?

Irritating global dependency: Solution

Introduce static setter

```
public class PermitRepository {
  private static PermitRepository instance = null;

  private PermitRepository() {}

  public static void setTestingInstance(PermitRepository newInstance) {
    instance = newInstance;
  }

  public static PermitRepository getInstance() {
    if (instance == null) {
      instance = new PermitRepository();
    }
    return instance;
  }
  ...
}
```

make it public, protected, package

Is it working?

Irritating global dependency: Solutions

Subclass and Override Method

```
public class TestingPermitRepository extends PermitRepository {
  ...
}
```

Extract Interface

```
public class PermitRepository implements IPermitRepository {
  private static IPermitRepository instance = null;

  protected PermitRepository() {}
  ...
}
```

I can’t run this method in a test harness

Common problems

The method is not accessible to the test

It is hard to call the method because it is hard to construct the parameters

The method has bad side effects (modifies a database, launches a cruise missile, etc.)

We need to sense through some objects used by the method

Hidden method

How do we write a test for a private method?

Two obvious solutions:

- Make the method public
- Change the private method to protected and then subclass it

What solution to choose?

- A matter of design choice for the original class…
- Private methods sometimes point to a SRP violation

Undetectable side-effects

```
public class AccountDetailFrame extends Frame implements ActionListener,
    WindowListener {
  private TextField display = new TextField(10);
  …
  private AccountDetailFrame(…) { … }

  public void performAction(ActionEvent event) {
    String source = (String)event.getActionCommand();
    if (source.equals("project activity")) {
      DetailFrame detailDisplay = new DetailFrame();
      detailDisplay.setDescription(getDetailText() + " " + getProjectText());
      detailDisplay.show();
      String accountDescription = detailDisplay.getAccountSymbol();
      accountDescription += ": ";
      …
      display.setText(accountDescription);
      …
    }
  }
}
```

Do you remember the example?

I need to make a change. What methods should I test?

Reasoning about effects

```
public class CppClass {
  private String name;
  private List declarations;
  public CppClass(String name, List declarations) {
    this.name = name; this.declarations = declarations;
  }
  public int getDeclarationCount() { return declarations.size(); }
  public String getName() { return name; }
  public Declaration getDeclaration(int index) {
    return ((Declaration)declarations.get(index));
  }
  public String getInterface(String interfaceName, int[] indices) {
    String result = "class " + interfaceName + " {\npublic:\n";
    for (int n = 0; n < indices.length; n++) {
      Declaration virtualFunction
        = (Declaration)(declarations.get(indices[n]));
      result += "\t" + virtualFunction.asAbstract() + "\n";
    }
    result += "};\n";
    return result;
  }
}
```

Things which can be changed that would affect results returned by any of CppClass methods?

Reasoning about effects: first results

Additional elements to declarations

- List is created first and passed to constructor
- List is held by reference
- Changes can alter the results of getInterface, getDeclaration, and getDeclarationCount

Alter or replace one of the objects held in declarations

- Affecting the same methods getInterface, getDeclaration, and getDeclarationCount

What about getName() and changes to name attribute?

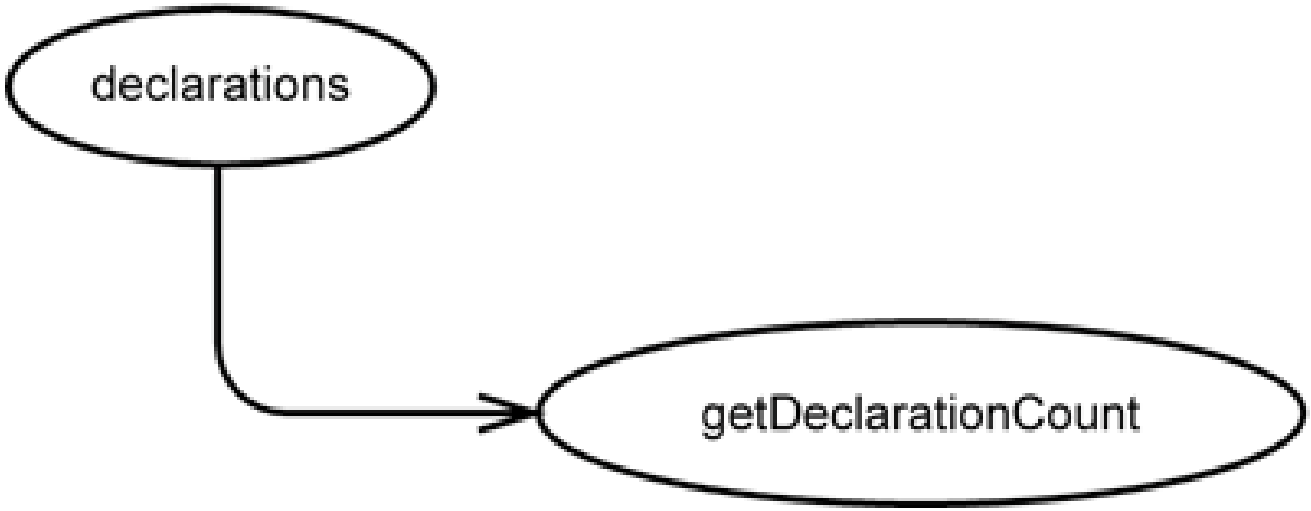
Effect sketch

Bubbles

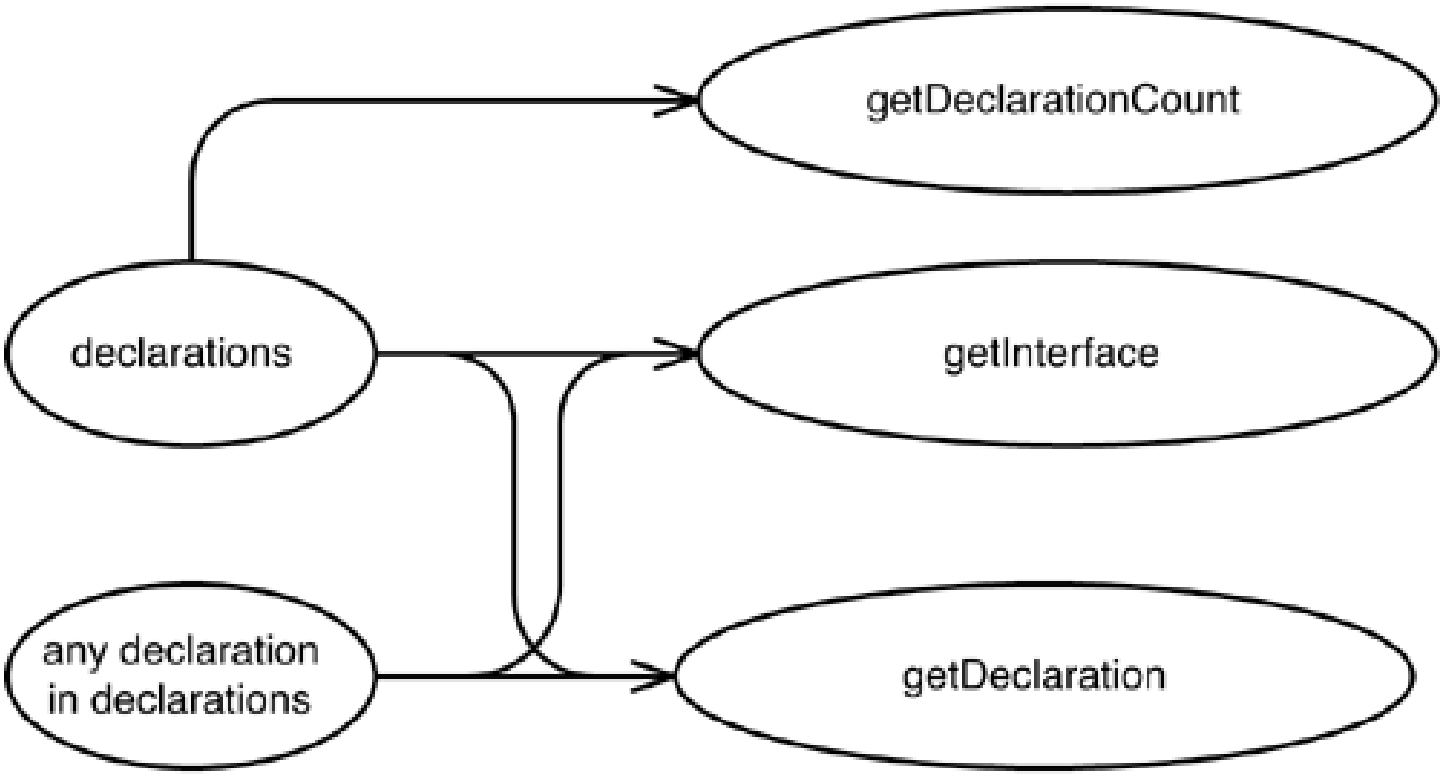
- Variables that can change
- Methods whose return values are affected

Arrows

- Depict the effect



Effect sketch of CppClass example



Reasoning forward

Directory index example

```
public class InMemoryDirectory {
  private List elements = new ArrayList<Element>();
  public void addElement(Element newElement) { elements.add(newElement); }

  public void generateIndex() {
    Element index = new Element("index");
    for (Element current : elements) {
      index.addText(current.getName() + "\n");
    }
    addElement(index);
  }

  public int getElementCount() { return elements.size(); }
  public Element getElement(String name) {
    for (Element current : elements) {
      if (current.getName().equals(name)) { return current; }
    }
    return null;
  }
}
```

If we call generateIndex twice?

Reasoning forward

Where are we going to make our changes?

```
public class InMemoryDirectory {
  private List elements = new ArrayList<Element>();
  public void addElement(Element newElement) { elements.add(newElement); }

  public void generateIndex() {
    Element index = new Element("index");
    for (Element current : elements) {
      index.addText(current.getName() + "\n");
    }
    addElement(index);
  }

  public int getElementCount() { return elements.size(); }
  public Element getElement(String name) {
    for (Element current : elements) {
      if (current.getName().equals(name)) { return current; }
    }
    return null;
  }
}
```

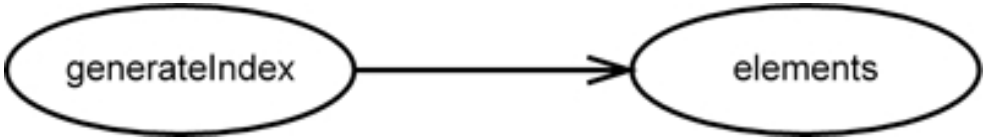
Add functionality

Remove functionality

What are the effects of these changes?

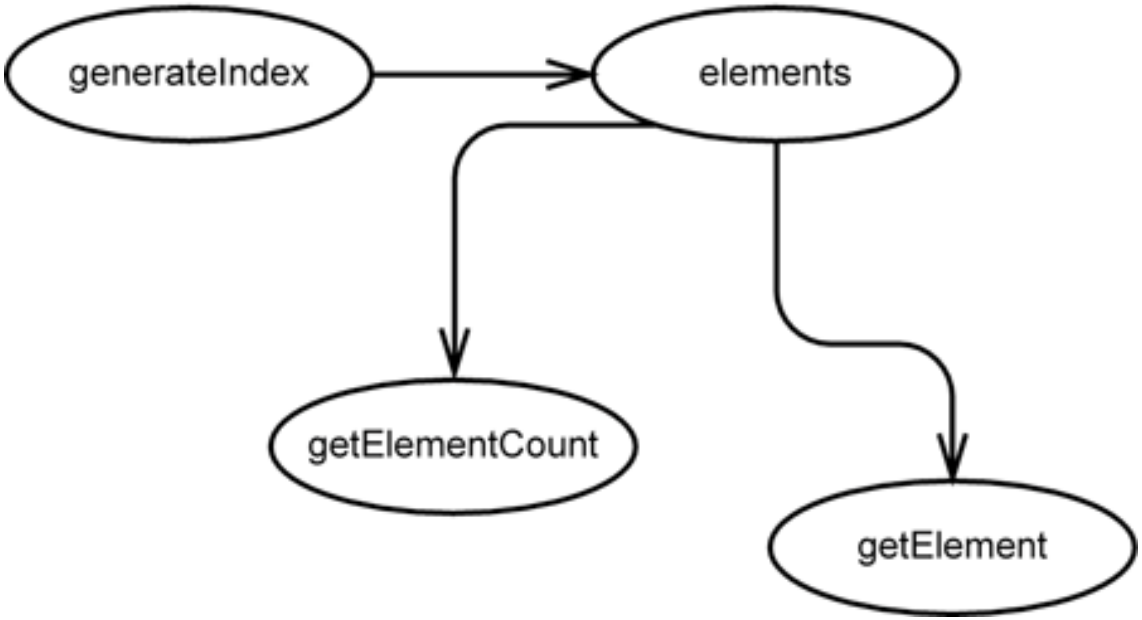
Reasoning forward: effect sketches (1)

Effect of changing functionality in generateIndex?

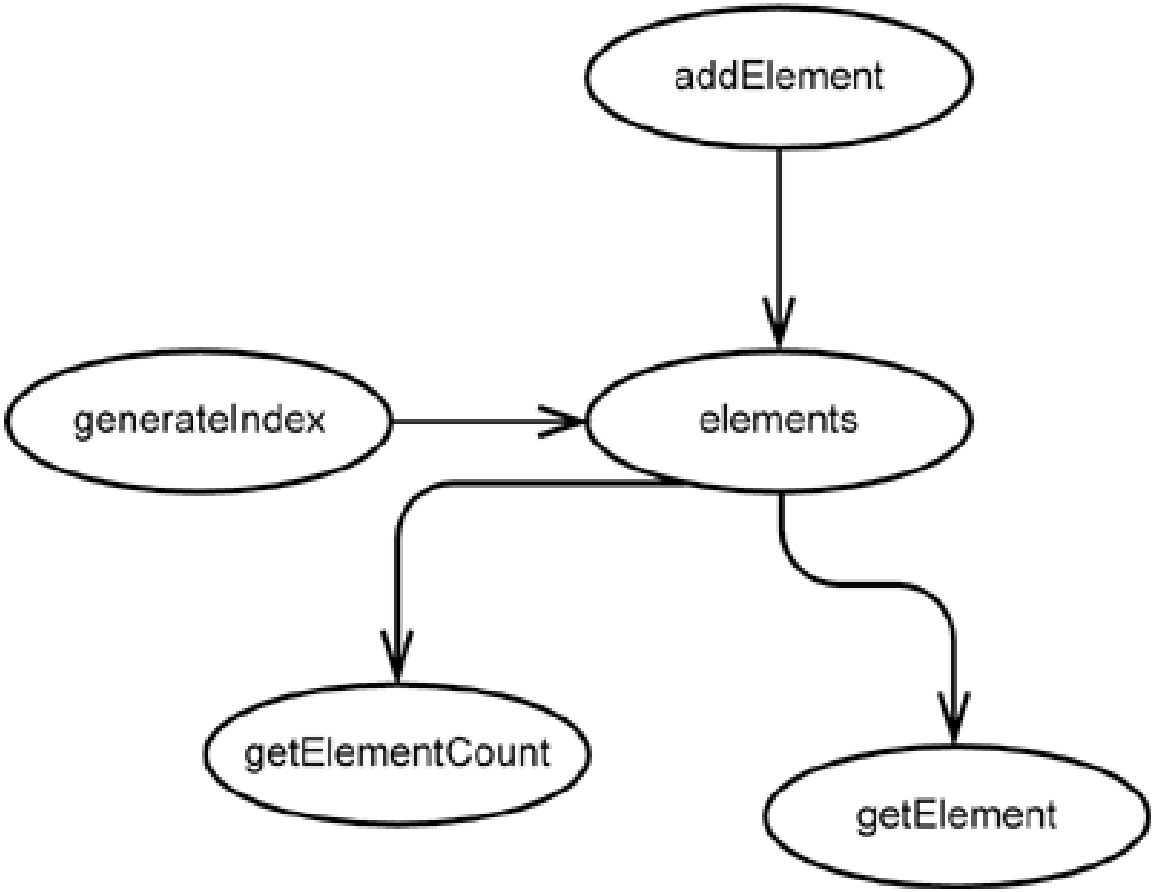


Reasoning forward: effect sketches (2)

Effect of changing elements?



Reasoning forward: effect sketches (3)



Make sure you have found all clients

Learning from effect analysis

Try to analyze effects in code whenever you get a chance

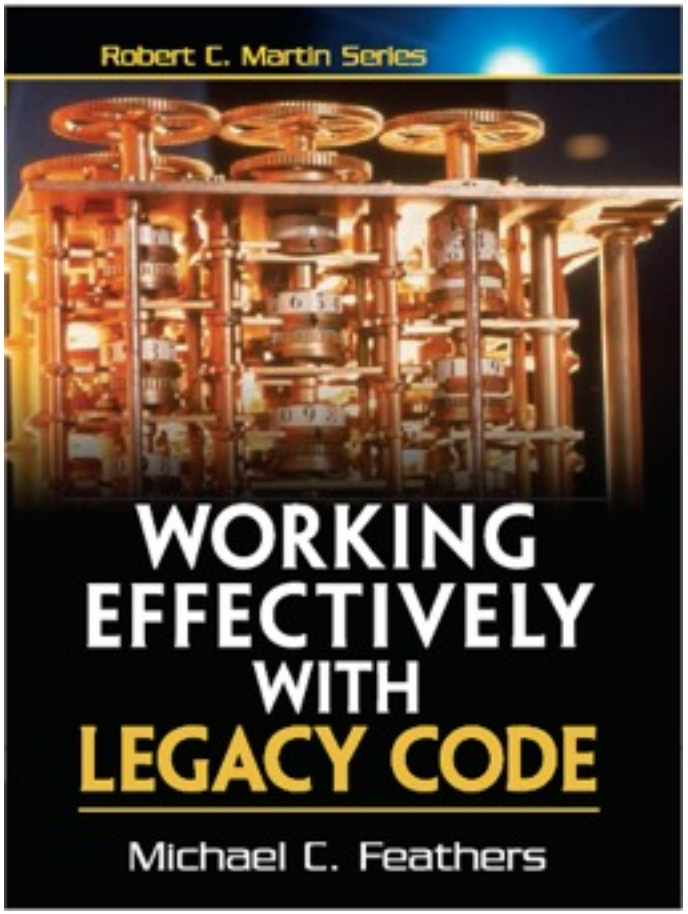
Try to narrow effects in a program

- Whenever you make a change you need less to understand the piece of code

Use tools

- Program slicing tools

More info and strategies



Working Effectively with Legacy Code
Michael Feathers, Prentice Hall, 1 edition, 2004

Conclusions

In order to reengineer, you need tests

- When creating tests, you might need to reengineer

Breaking dependencies help to introduce tests

Analyze effects of changes

Lean on tools

- Be careful with manual refactorings