Reengineering I

Slides are based on the reengineering lecture

Legacy Software System

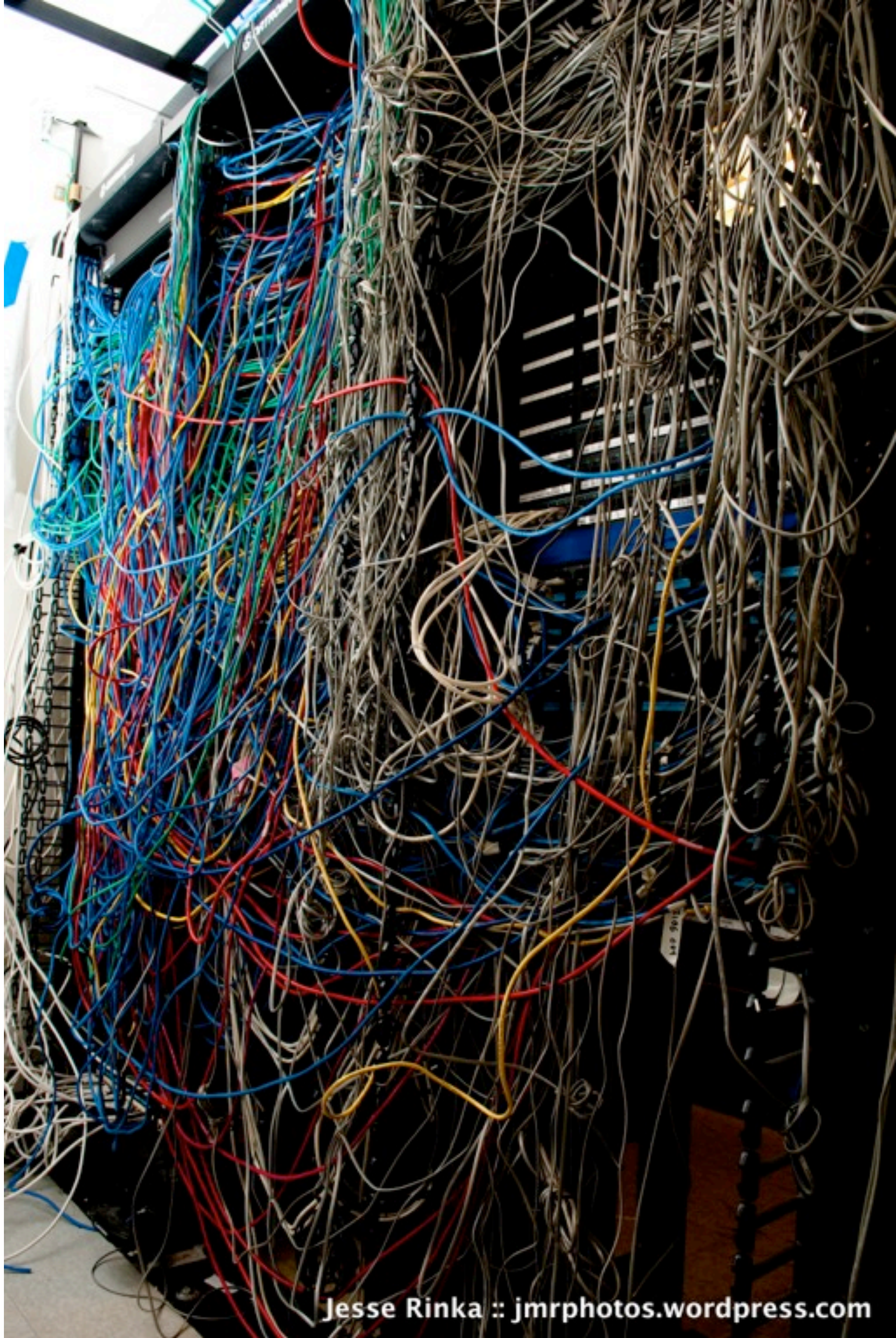
- Long running, existing business software is called *legacy software*
- Often business critical
- In most cases it runs and works properly
- Huge amount of money and time has been invested
- A lot of experience with such an existing system
- After all: *Never ever changes a running system*

Lehman’s Laws

- We must add new features to existing systems
- What shall we do with our legacy system?
- Re-Implement and replace it with the new features?
- Extend the existing legacy software with new features?
- The problem: In practice new features are often added in *ad hoc manner*
- Copy-Past Programming, if-branches, adding dependencies, ...
- No tests, no documentation updates, ...

The Result

- System becomes one large inter-wired monolith
- Try to change *only one* of these dependencies!
- How sure can you be not to *break anything* else?
- Those sloppy, non structured ad-hoc changes are what drives Lehman's Laws in practice



Vicious Circle

The more complex a system is the more tempting are ad-hoc changes

Ad-hoc Changes

- The seem to be less costly and faster to implement *at first*
- Do a *quick hack* - it’s only a matter of hours
- They even do their job (for some time)
- This is not a *sustainable* point of view
- Ad-hoc changes do not follow the original architecture
- It a short-term mindset, but any legacy system runs over a long time
- Ad-hoc changes are like taking a *loan* or *credit*
- There is going to be pay back - with huge interests rates

(Mid- to Long-Term) Consequences 1

- Changing one module leads to a rat-tail of changes in other modules: Even small changes take too much time and are complex
- More defects
- Bad time to market performance (changes take longer to implement)
- Evolution of the system controls us: The only way to implement changes are ad-hoc changes
- We spend most of the resources dealing with problems that were caused by prior ad-hoc modifications: Once a single defect is fixed, new ones pop up
- At some point: We are hardly implementing any new features

(Mid- to Long-Term) Consequences 2

- Documentation is out of synch with real system
- No tests (there are no well defined interfaces anymore to be tested)
- Due to all the (circular) dependencies, building the system is difficult and results in long compilation times.
- The original, clean design and the architecture of the system are long gone
- Only the most senior developers know the system (and how and where to change it)
- We have no choice but to implement new requirements by ad-hoc changes

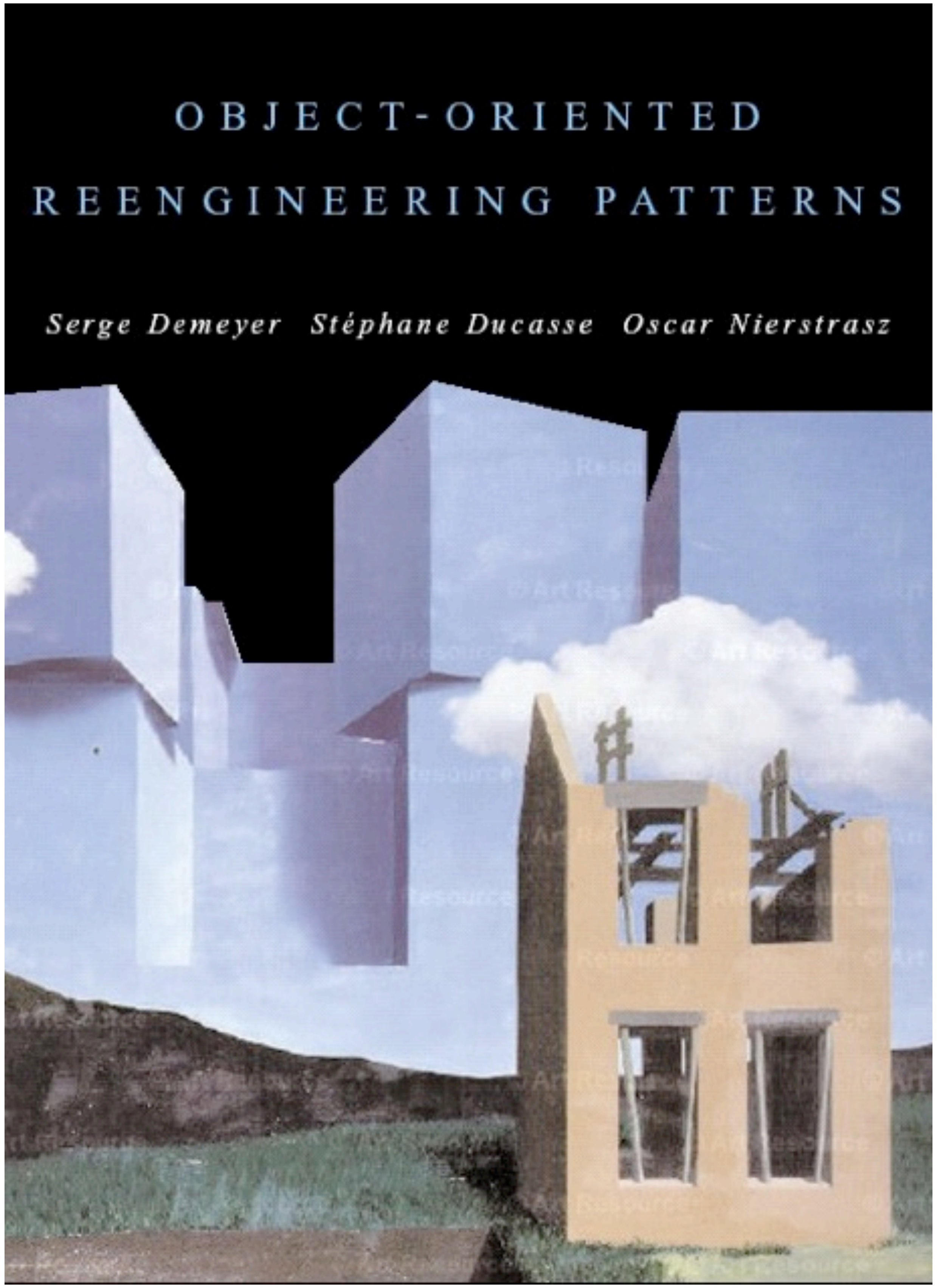
In the long run we are all dead!

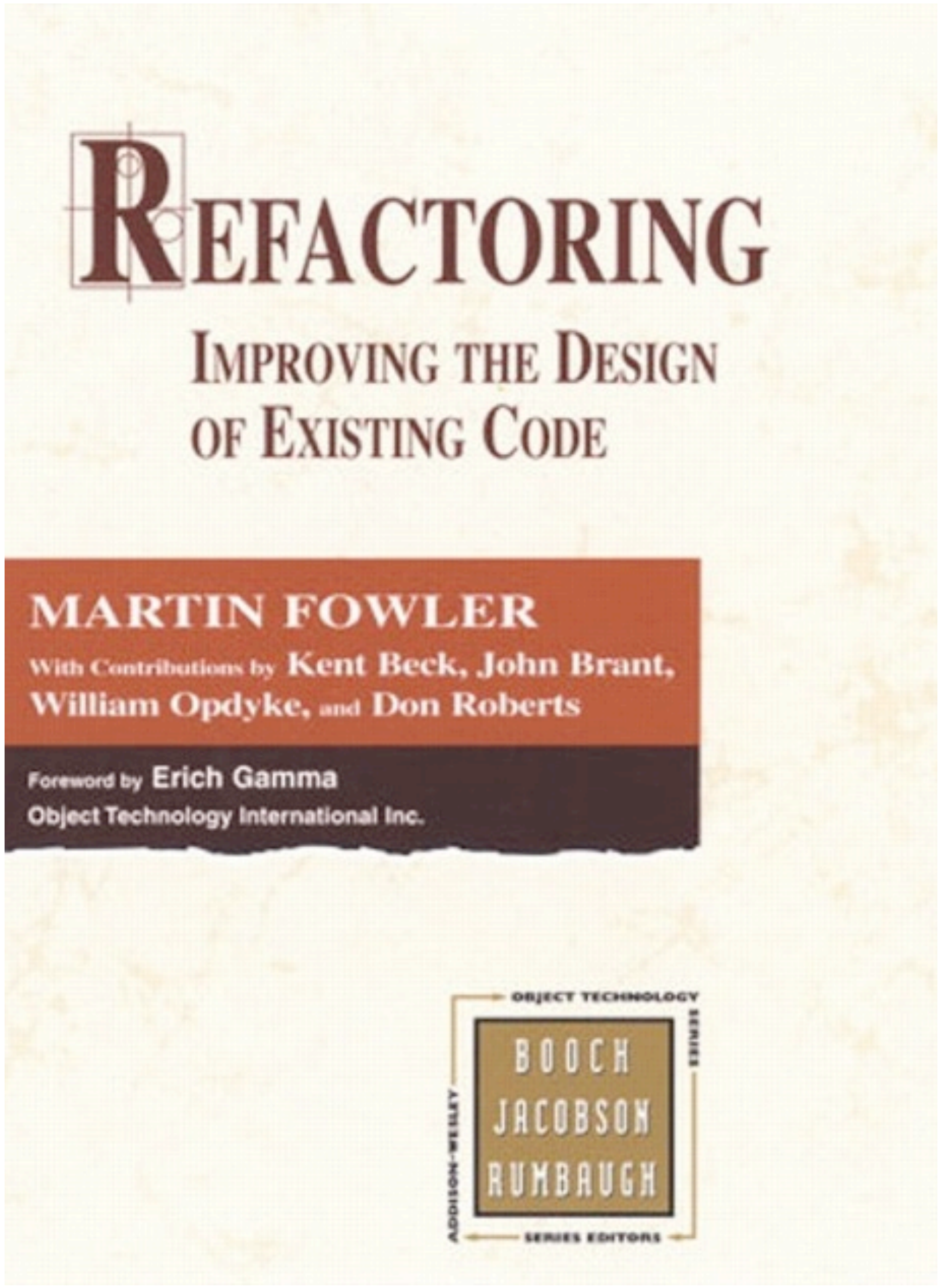
John Maynard Keynes

Reengineering can help

- ... when the evolution of a system gets out of control
- ... to break the vicious circle of increasing complexity and ad-hoc changes
- Reengineering is concerned with restructuring a system [Demeyer et al.]
- Prepare a software system for future development and new requirements without the use of ad-hoc changes

“*Reengineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.*”
[Demeyer et al.]



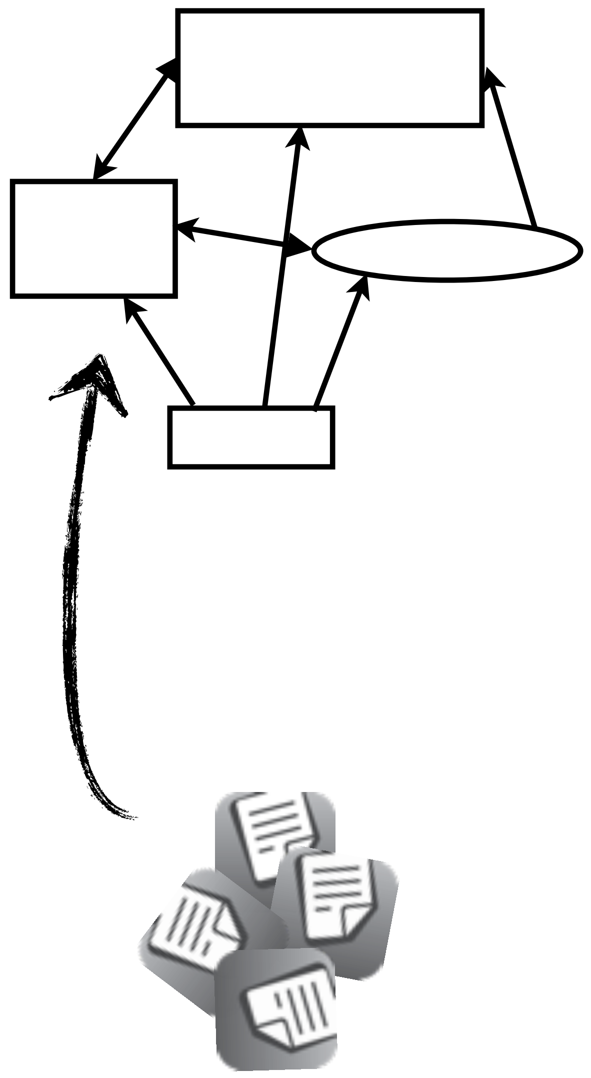

REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International Inc.
OBJECT TECHNOLOGY SERIES
BOOCH JACOBSON RUMBAUGH
SERIES EDITORS
ADDISON-WESLEY

Object-Oriented Reegineering Patterns *by*
S. Demeyer, S. Ducasse, and O. Nierstrasz
free copy @ http://scg.unibe.ch/download/oorp/

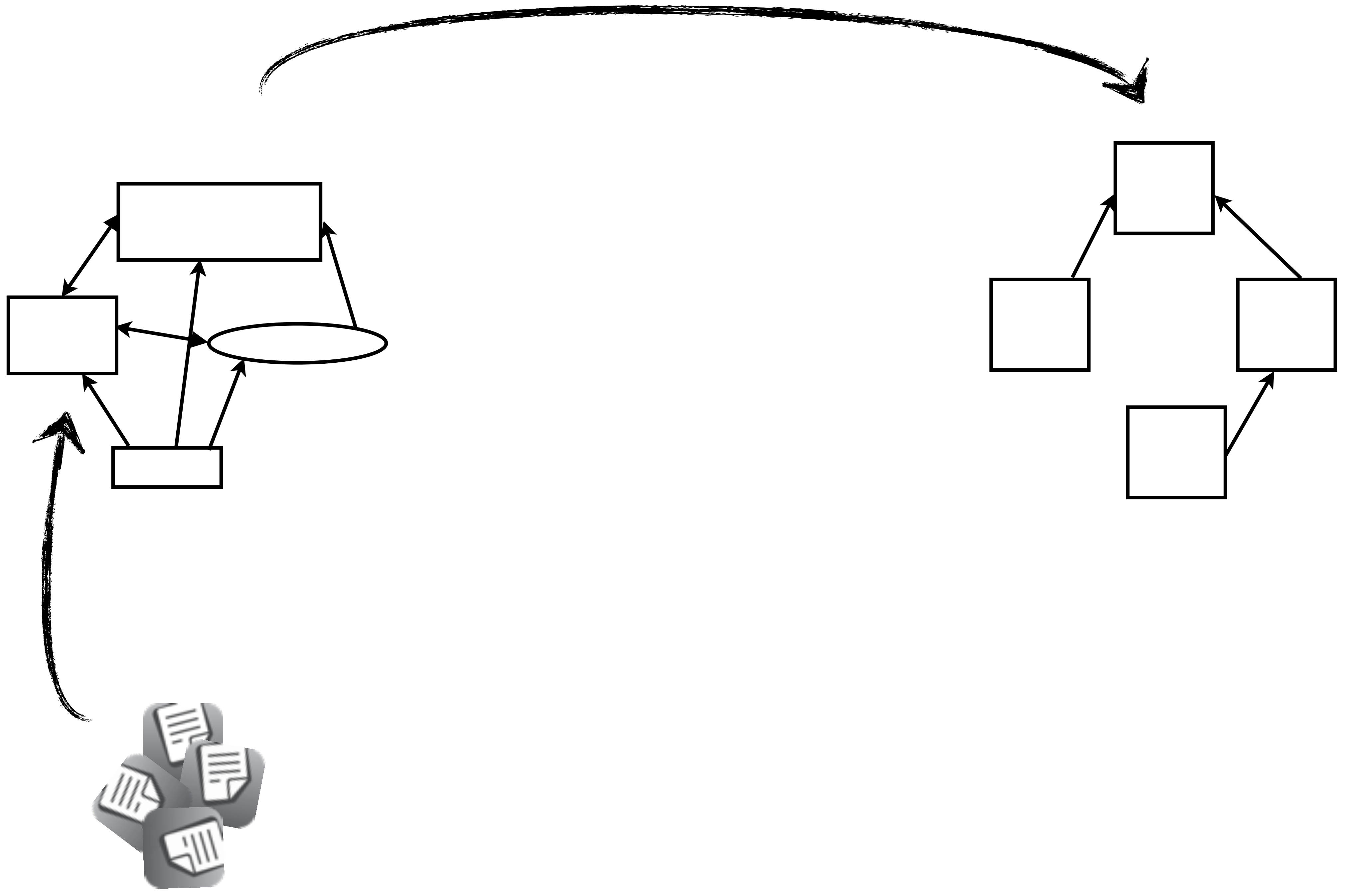
Refactoring: Improving the Design of existing Code *by*
Martin Fowler
Addision-Wesley Professional , 1999

Reengineering Workflow

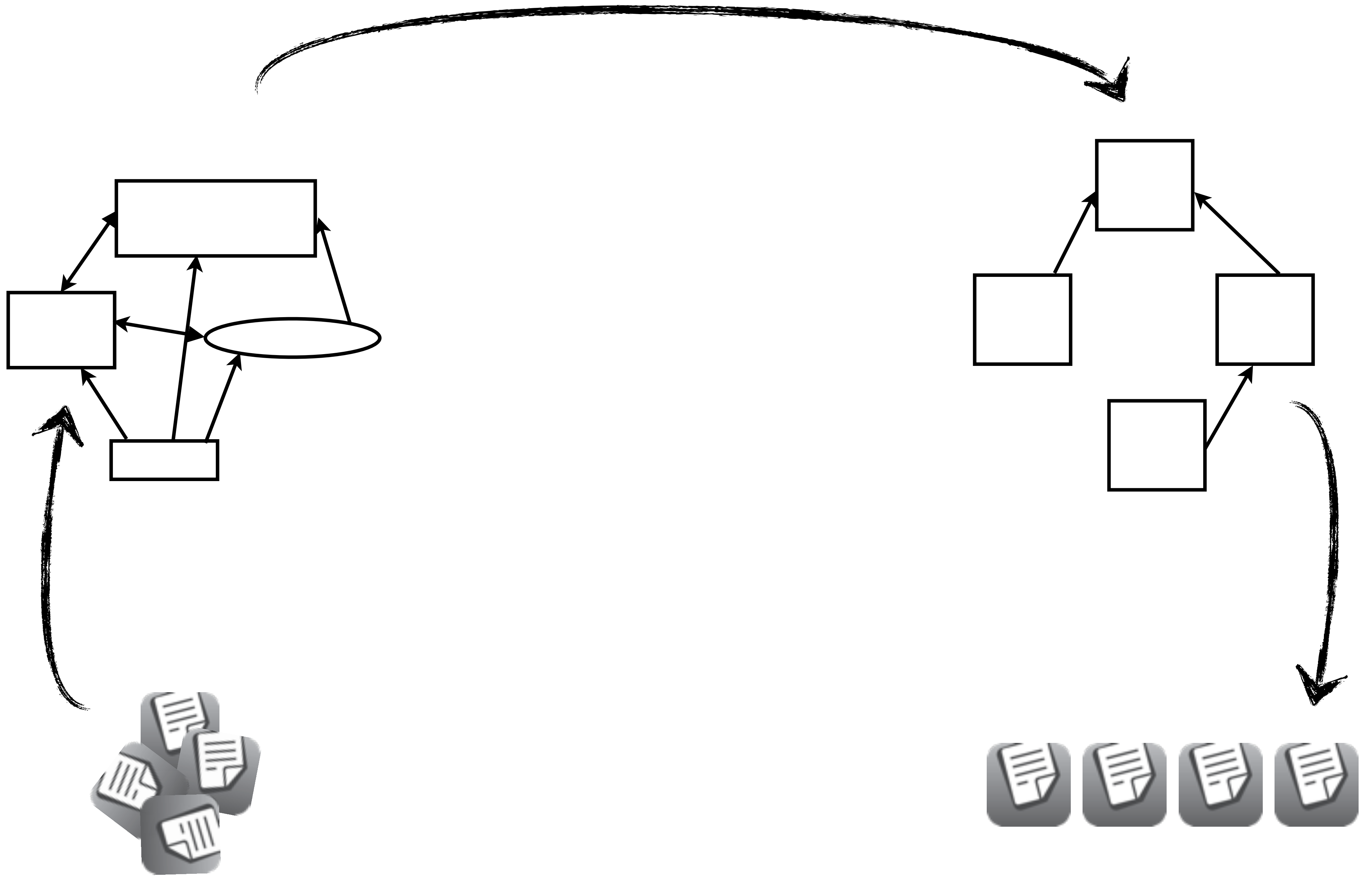
Reengineering Workflow



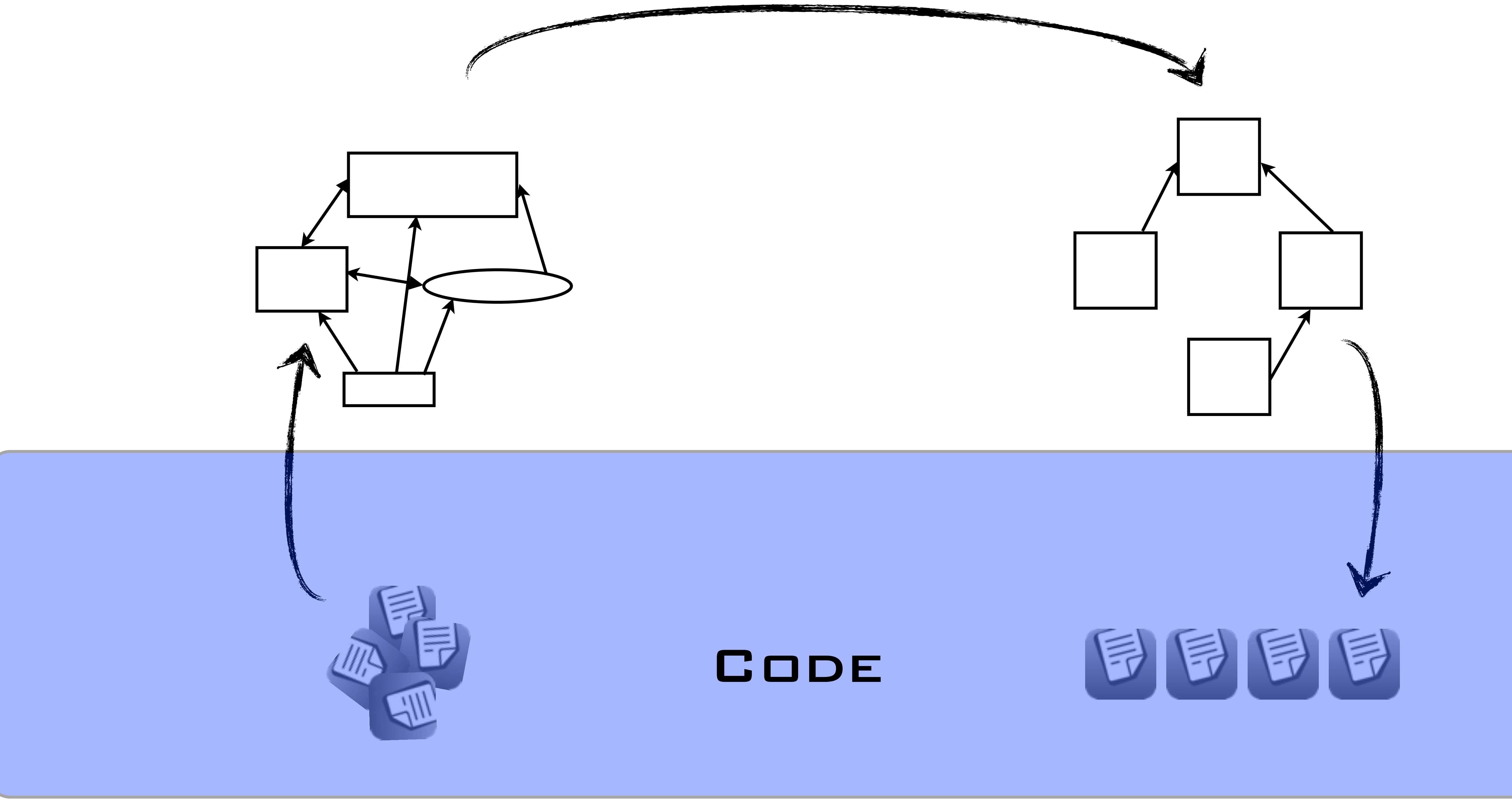
Reengineering Workflow



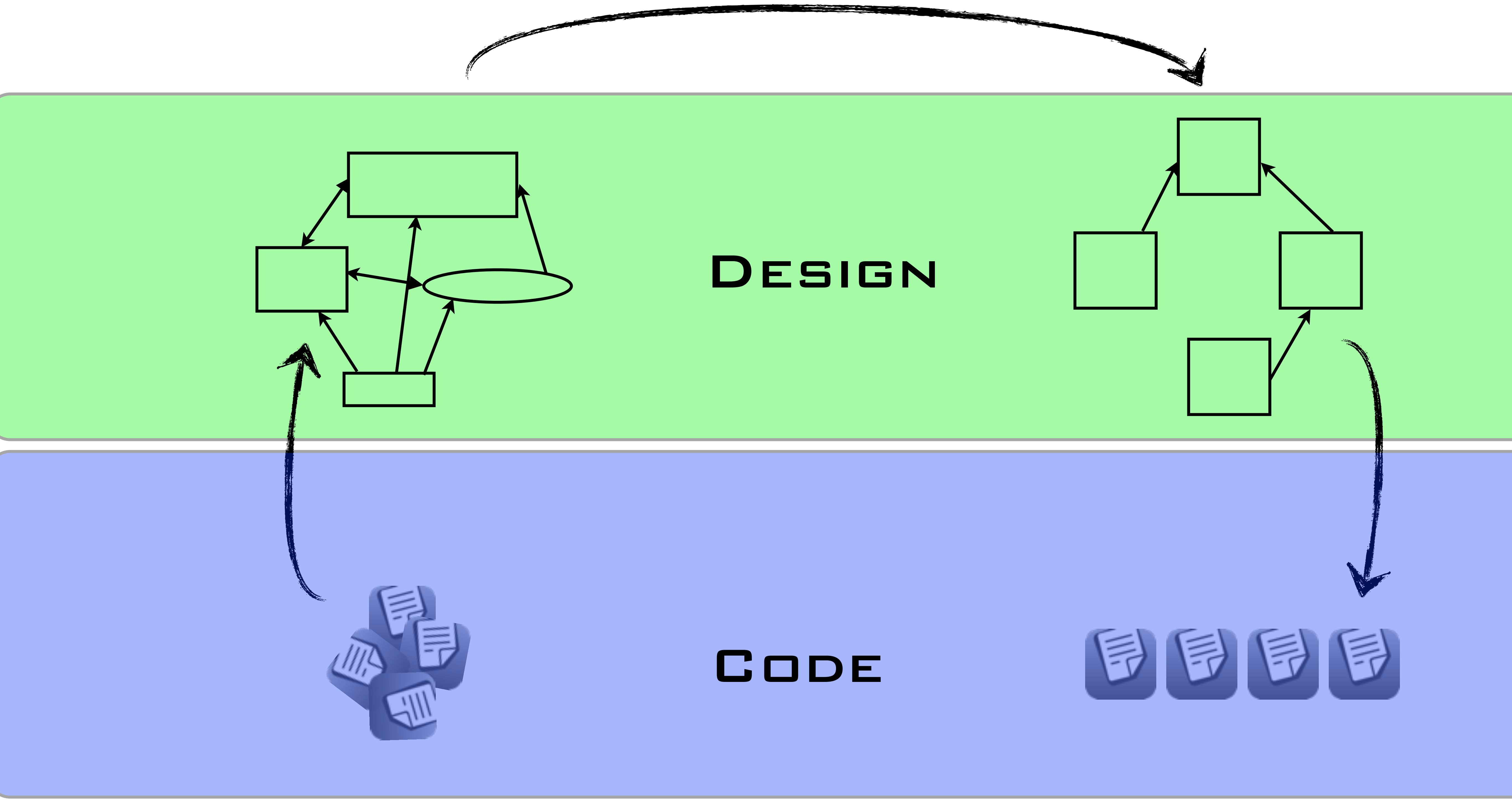
Reengineering Workflow



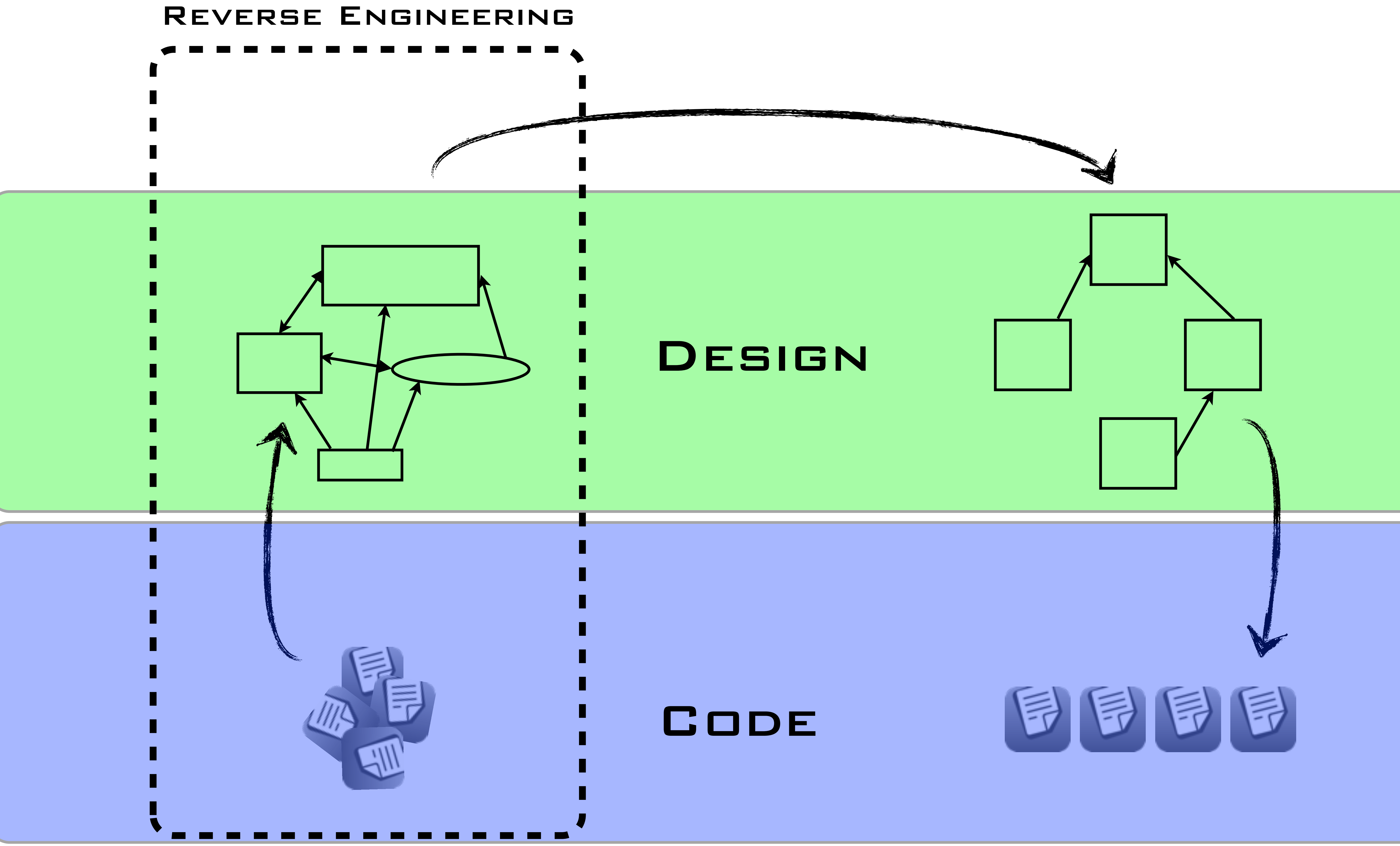
Reengineering Workflow



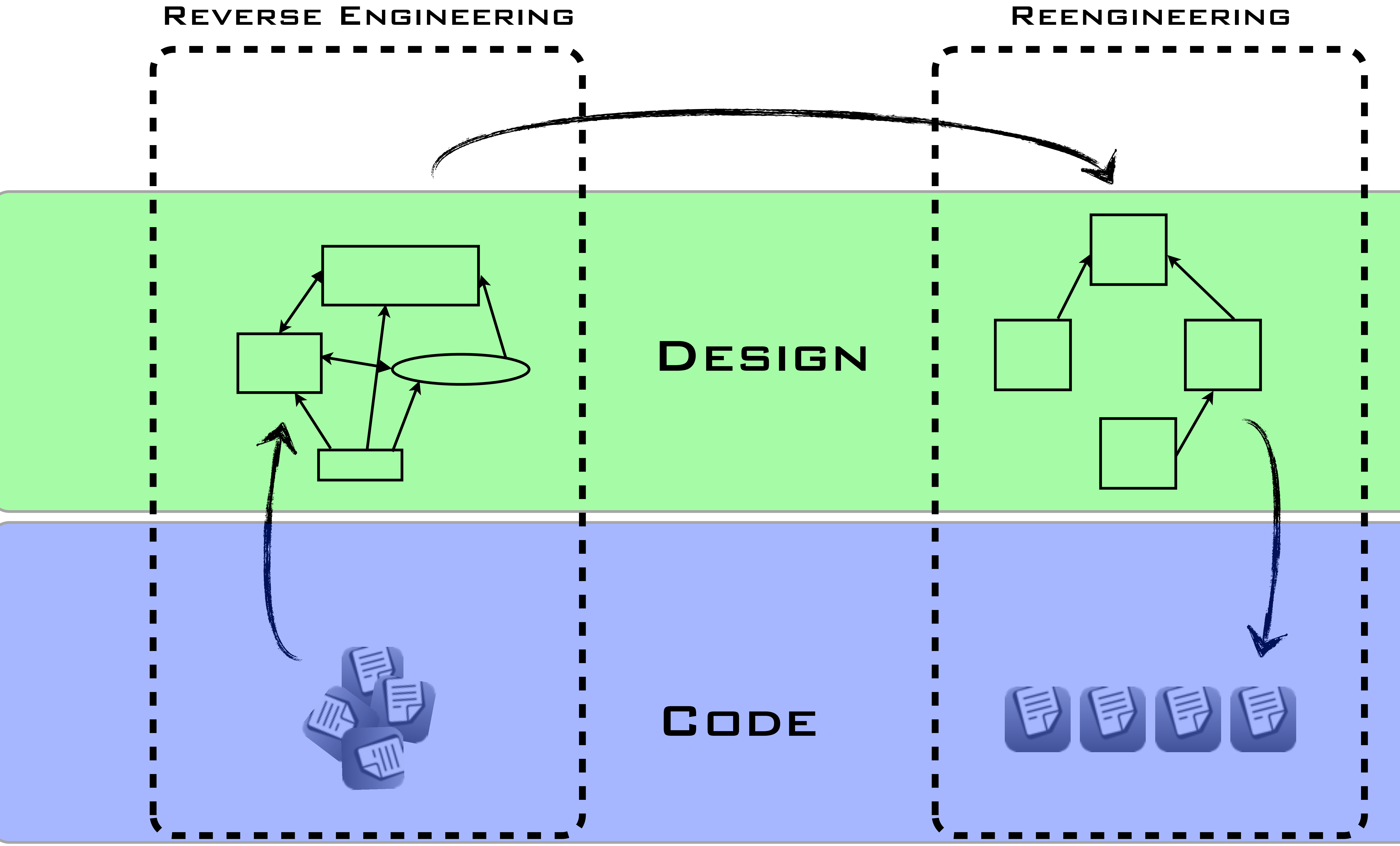
Reengineering Workflow



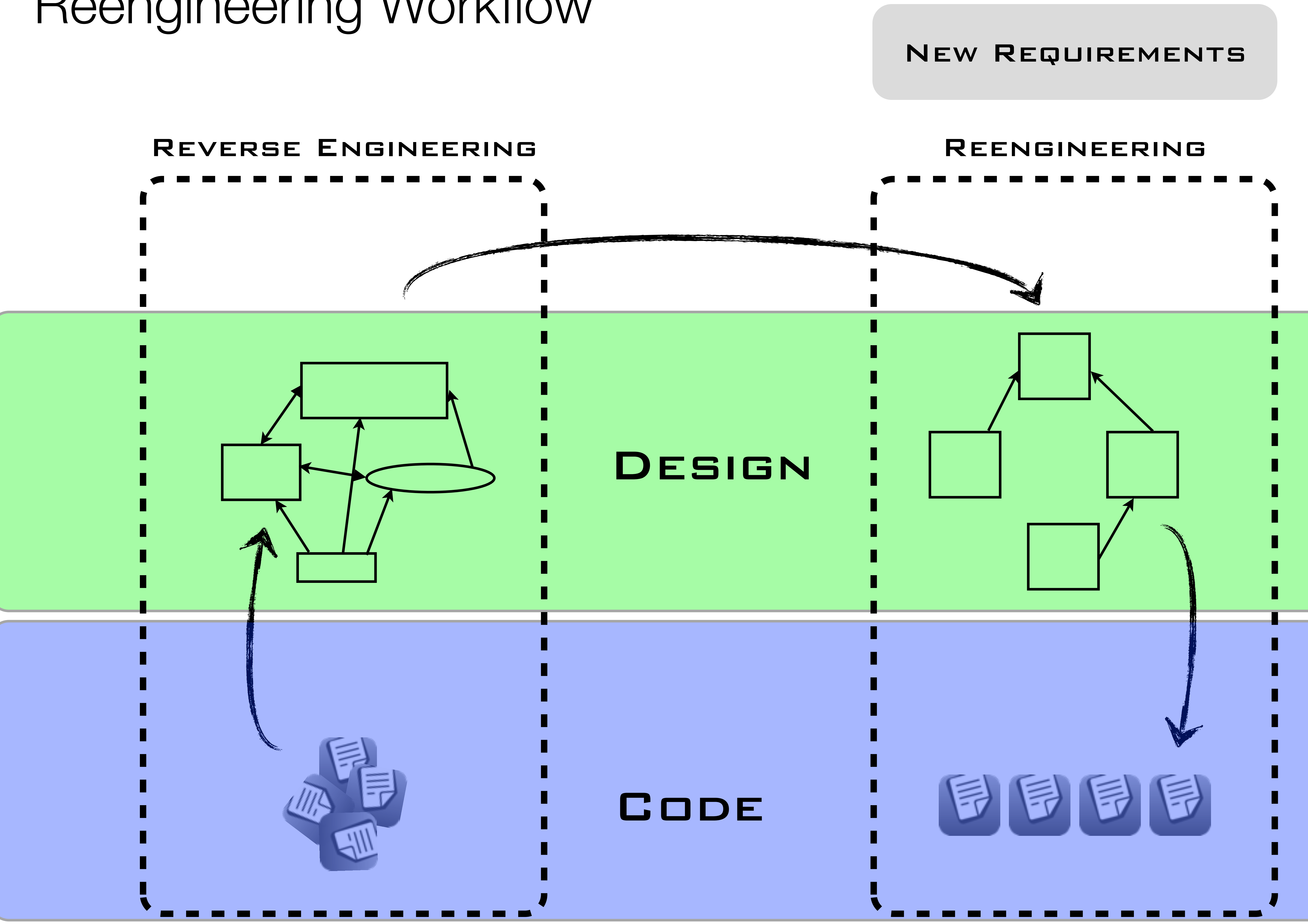
Reengineering Workflow

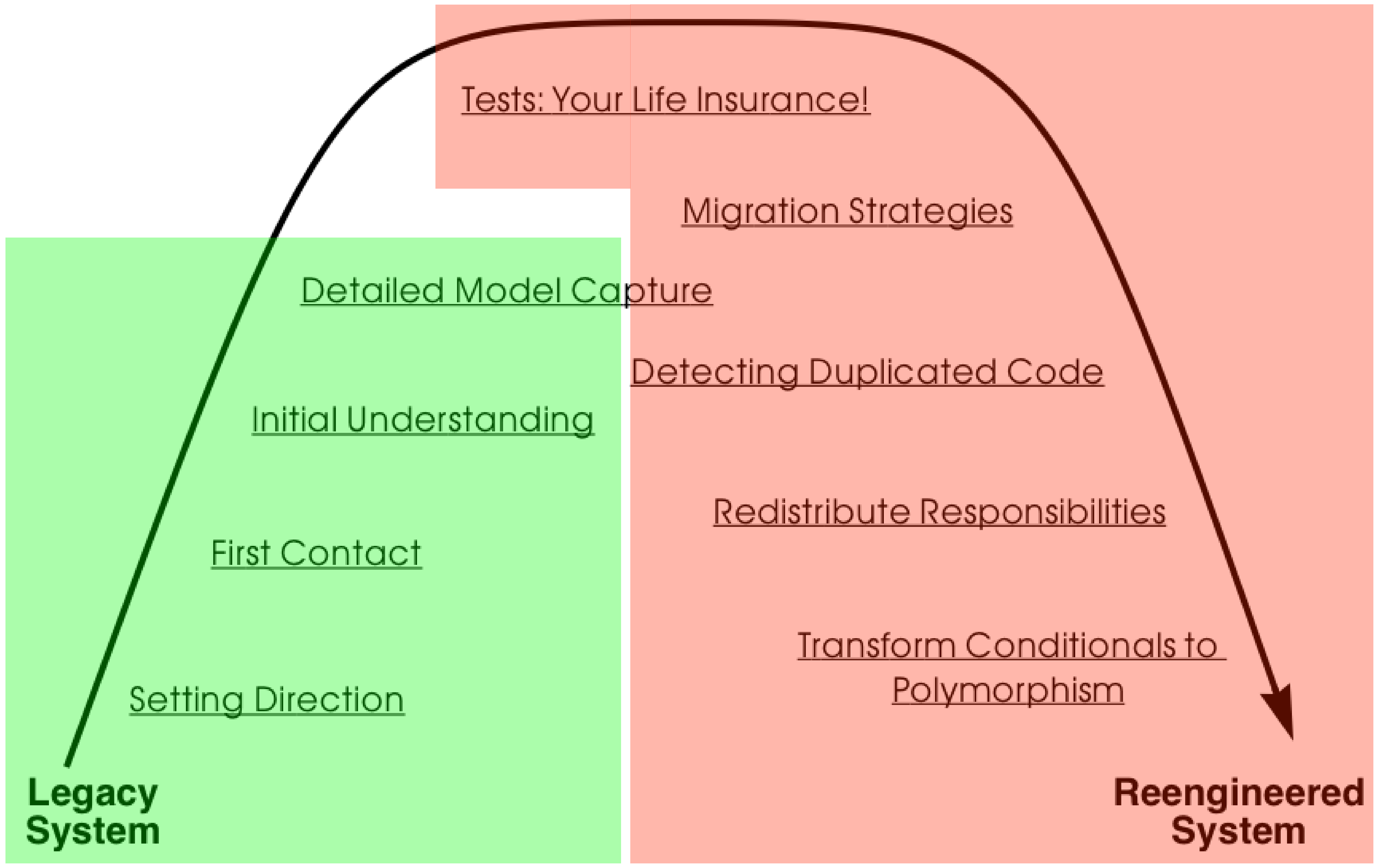


Reengineering Workflow



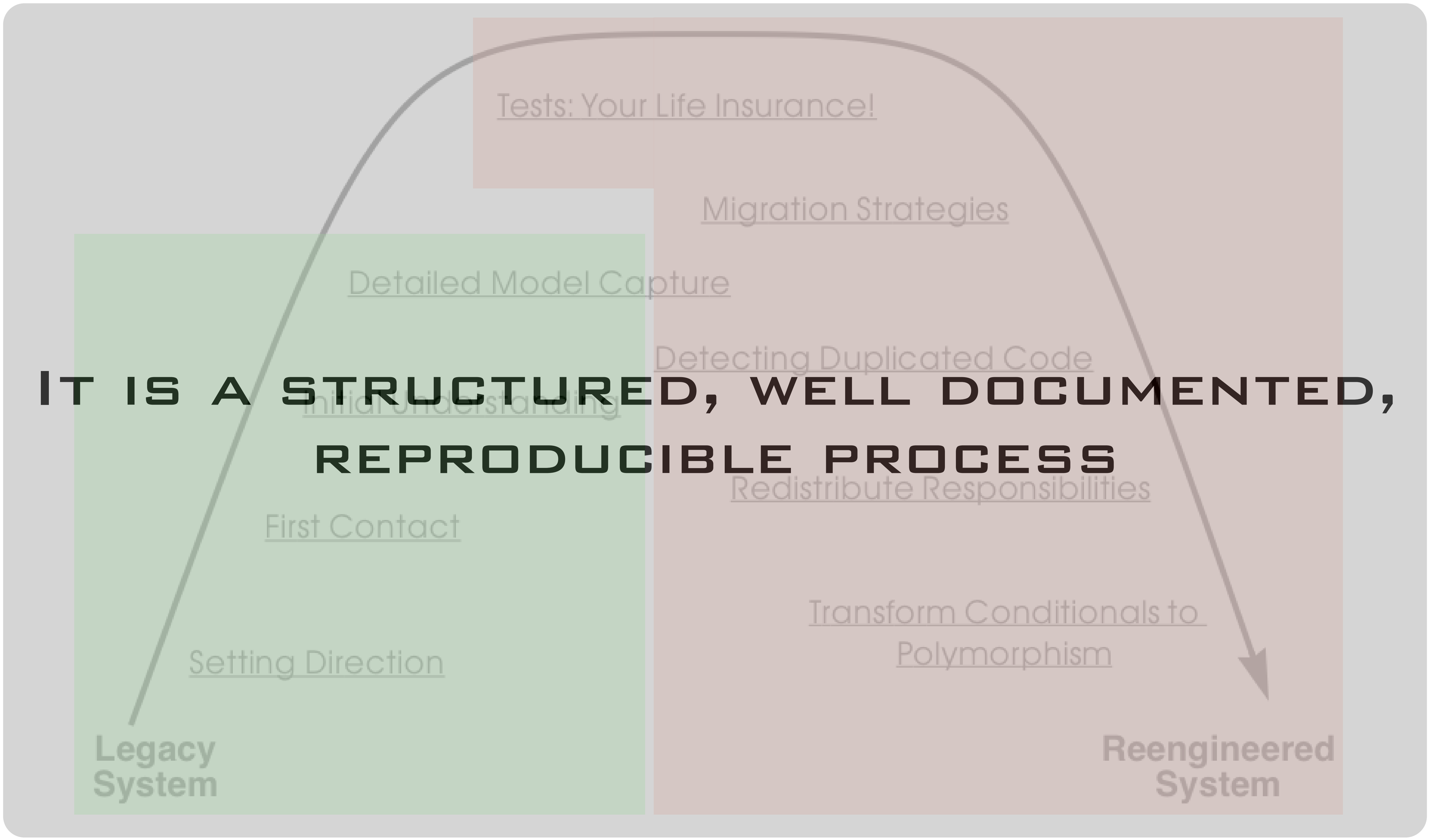
Reengineering Workflow





Reverse- and Reengineering are step wise procedures

Demeyer et al.



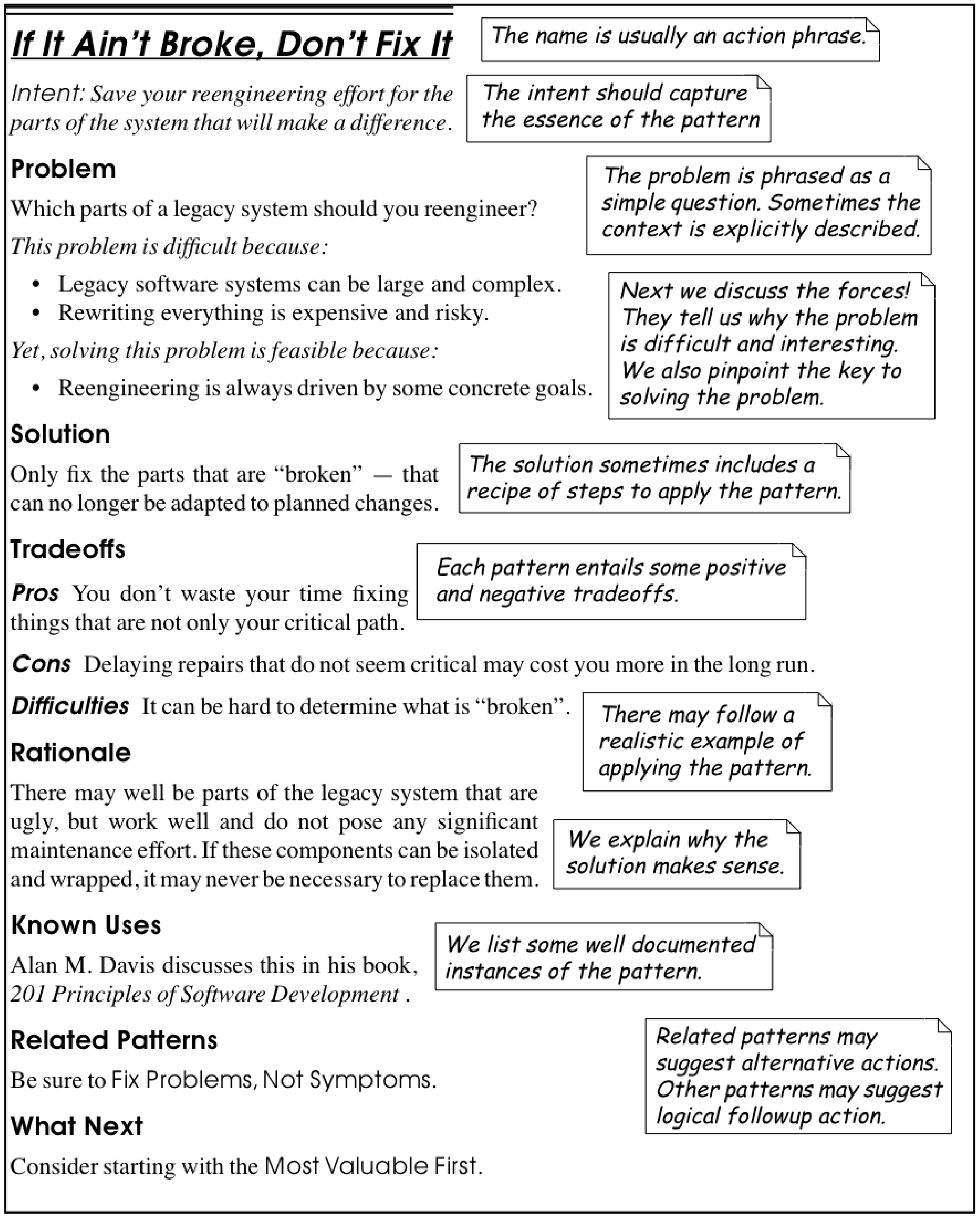
IT IS A STRUCTURED, WELL DOCUMENTED, REPRODUCIBLE PROCESS

Reverse- and Reengineering are step wise procedures

Demeyer et al.

Reengineering Pattern Catalogue

- Description how to solve a problem (in the reengineering context)
- Capture best practices how to solve the problem
- Not necessarily “new”, but formalization of existing knowledge
- Use of a common vocabulary
- Relation between patterns



If It Ain’t Broke, Don’t Fix It

The name is usually an action phrase.

Intent: *Save your reengineering effort for the parts of the system that will make a difference.*

The intent should capture the essence of the pattern

Problem

Which parts of a legacy system should you reengineer?

The problem is phrased as a simple question. Sometimes the context is explicitly described.

This problem is difficult because:

- Legacy software systems can be large and complex.
- Rewriting everything is expensive and risky.

Yet, solving this problem is feasible because:

- Reengineering is always driven by some concrete goals.

Next we discuss the forces! They tell us why the problem is difficult and interesting. We also pinpoint the key to solving the problem.

Solution

Only fix the parts that are “broken” — that can no longer be adapted to planned changes.

The solution sometimes includes a recipe of steps to apply the pattern.

Tradeoffs

Each pattern entails some positive and negative tradeoffs.

Pros You don’t waste your time fixing things that are not only your critical path.

Cons Delaying repairs that do not seem critical may cost you more in the long run.

Difficulties It can be hard to determine what is “broken”.

There may follow a realistic example of applying the pattern.

Rationale

There may well be parts of the legacy system that are ugly, but work well and do not pose any significant maintenance effort. If these components can be isolated and wrapped, it may never be necessary to replace them.

We explain why the solution makes sense.

Known Uses

Alan M. Davis discusses this in his book, *201 Principles of Software Development* .

We list some well documented instances of the pattern.

Related Patterns

Be sure to Fix Problems, Not Symptoms.

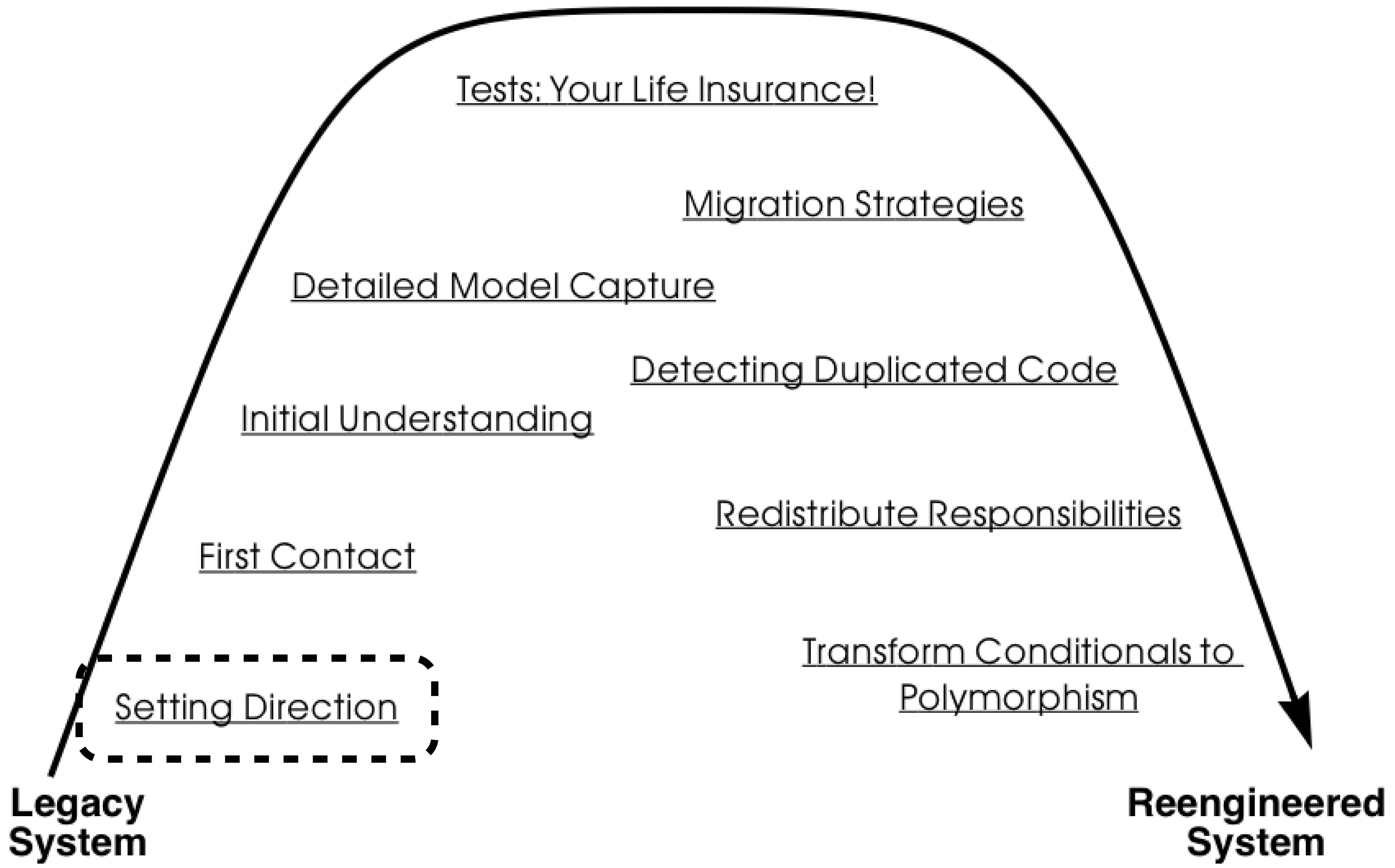
Related patterns may suggest alternative actions. Other patterns may suggest logical followup action.

What Next

Consider starting with the Most Valuable First.

Reverse Engineering Recap

- The goal of reverse engineering is *understanding how an existing system "works"*
    - identify (important parts) of the system
    - understand dependencies
    - building an abstract model of the system
- We need to understand the system because we can not reengineer and transform what we do not understand
- Transforming *without* a solid *understanding* would be *ad-hoc development*



Setting Direction (Ch. 2)

Don’t set the stakes too high

Setting Direction (Chapter 2)

- It is about getting a *focused mindset* before embarking the actual reengineering journey

- Reengineering is a holistic approach with (too) many dimensions

1001 PROBLEMS

1001 WAYS TO DO IT WRONG

1001 OPTIONS

1001 WAYS TO DO IT RIGHT

1001 STARTING POINTS

1001 OPINIONS

1001 PROBLEMS

1001 WAYS TO DO IT WRONG

1001 OPTIONS

1001 WAYS TO DO IT RIGHT

1001 STARTING POINTS

1001 OPINIONS

It is easy to get lost before you actually start

Setting Direction (Chapter 2)

- It is about getting a *focused mindset* before embarking the actual reengineering journey
- Reengineering is a holistic approach with (too) many dimensions
- Decide where you start, what to do, how to do it
- Once the decision is made, stick to that plan

A Few Patterns

- Most valuable first
  - Some problems are critical, other problems are minor issues
  - Find out what is important for the customers
  - Can be difficult (Usage statistics, log files, business models)
- Keep it simple
- Don’t fix it if it ain’t broken
- The patterns are rather general and apply to other kind of projects too

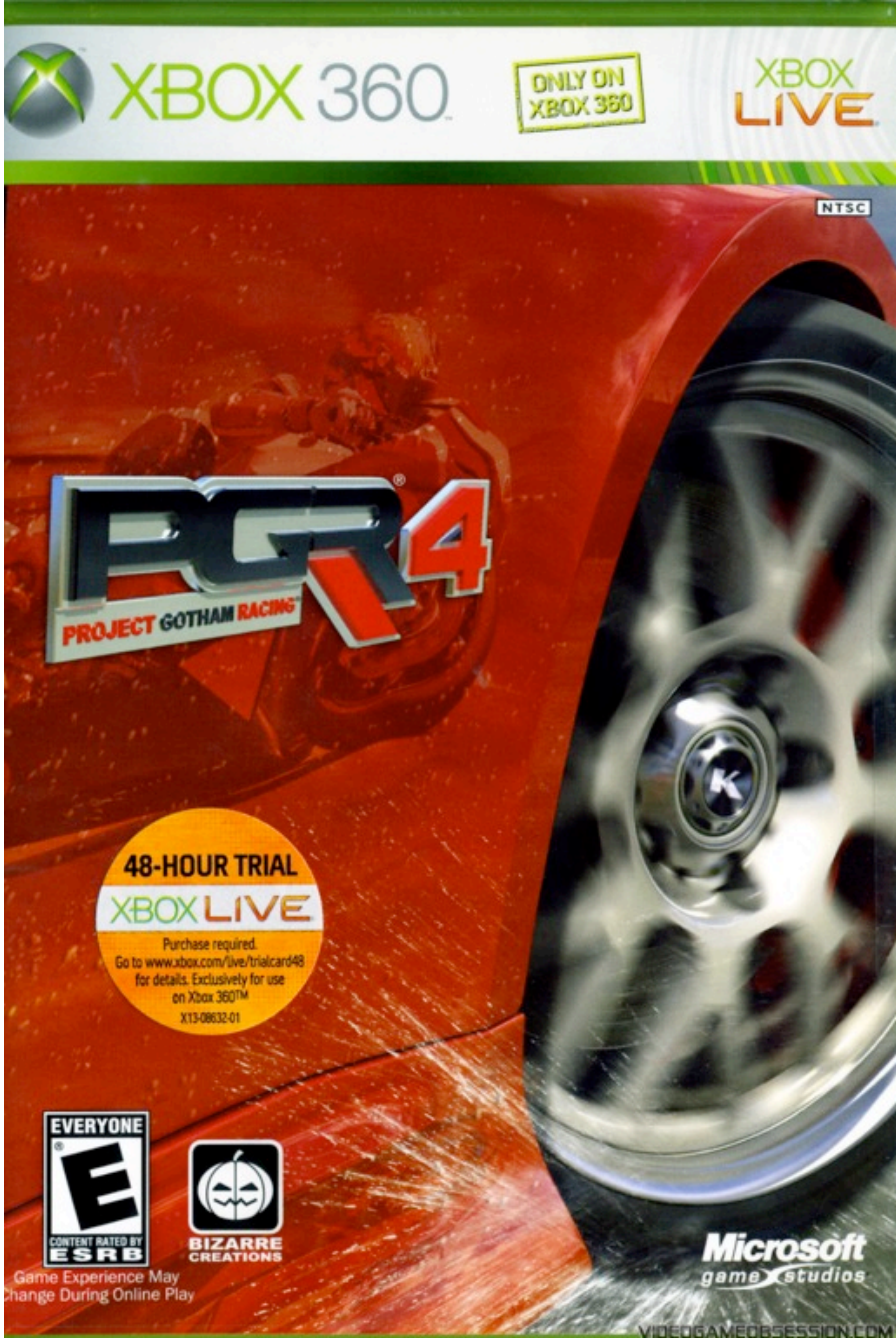
DRAGON AGE™
ORIGINS

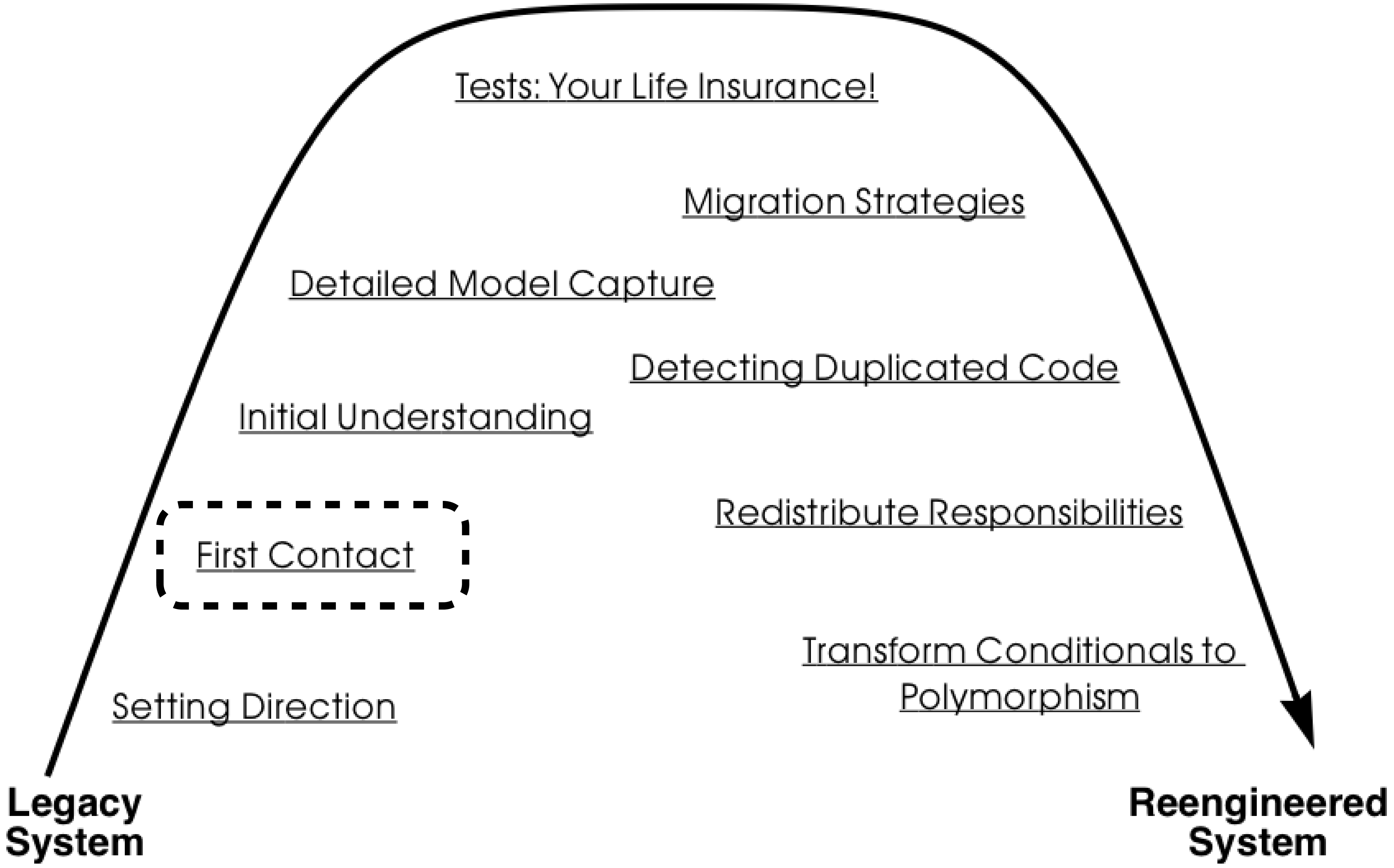
Visualizing and Understanding Players' Behavior in Video Games: Discovering Patterns and Supporting Aggregation and Comparison
by Dinara Moura et al. @ Game Track, SIGGRAPH'11, Vancouver

BIOWARE®

Data Analytics for Game Development by Hullet et al. @ NIER-Track, ICSE'11

- 53 of 133 vehicles used in < 0.25% of races
- 7 of 16 event types were used in < 0.1% of races
- 2 of 4 game modes were used in < 2% of races



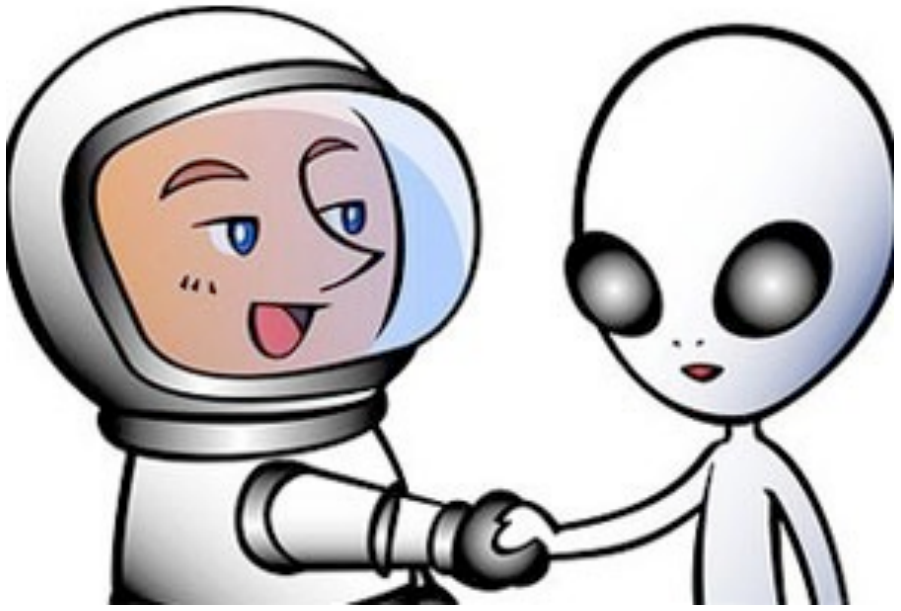


First Contact (Ch. 3)

Get a quick overview of the situation

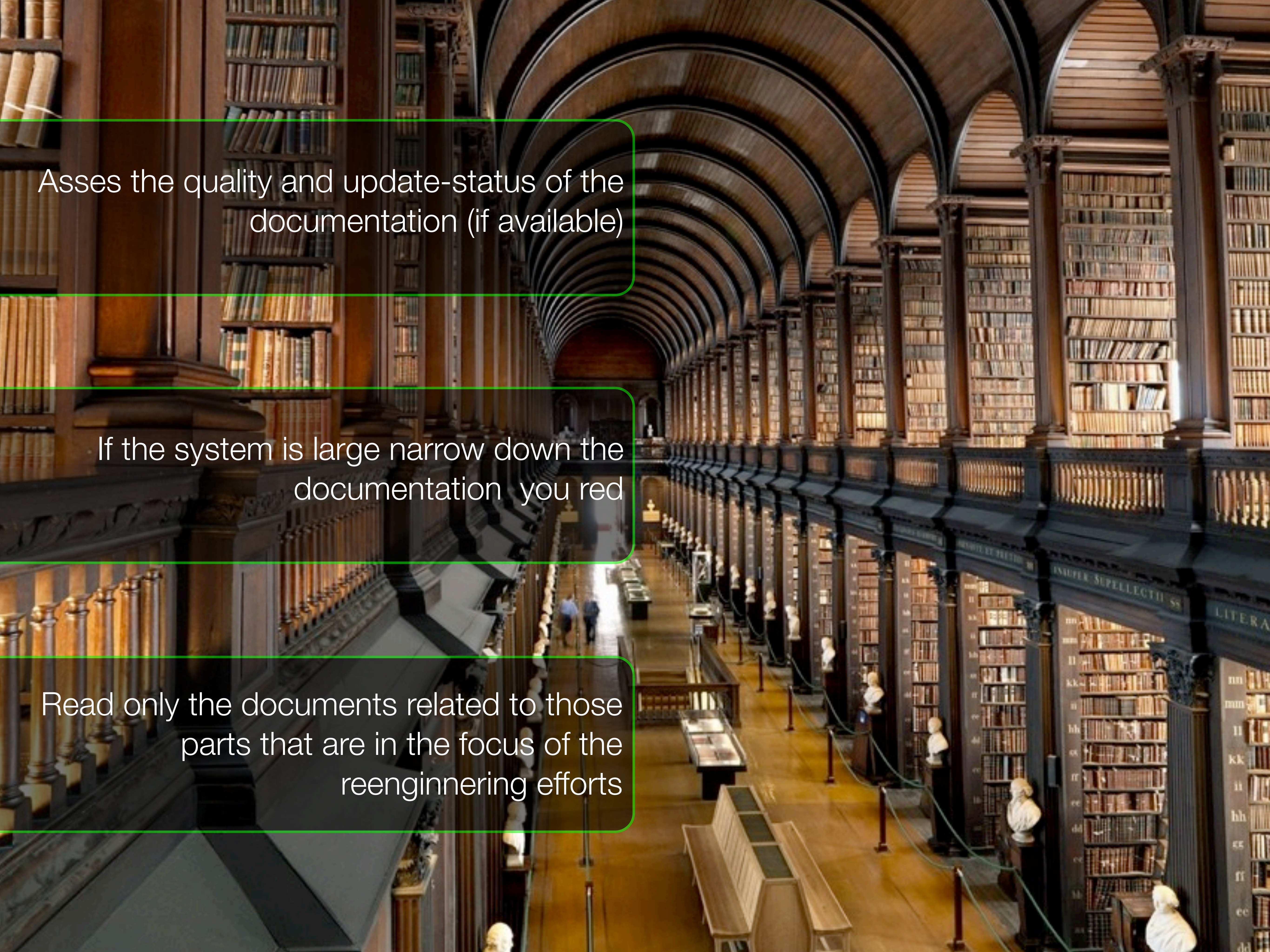
First Contact (Chapter 3)

- Get a *first quick overview* of the *current state* of the system
- Conduct a first assessment of the feasibility of the reengineering project
- Grasp the main issues, identify risks, and opportunities
- First contact is critical because it can trigger some initial decisions
- Time critical



A Few Patterns

- Chat with the maintainers
- Skim the documentation



Asses the quality and update-status of the documentation (if available)

If the system is large narrow down the documentation you red

Read only the documents related to those parts that are in the focus of the reenginnering efforts

An assessment of the usefulness of documents in the focus and context of the reengineering project

A Few Patterns

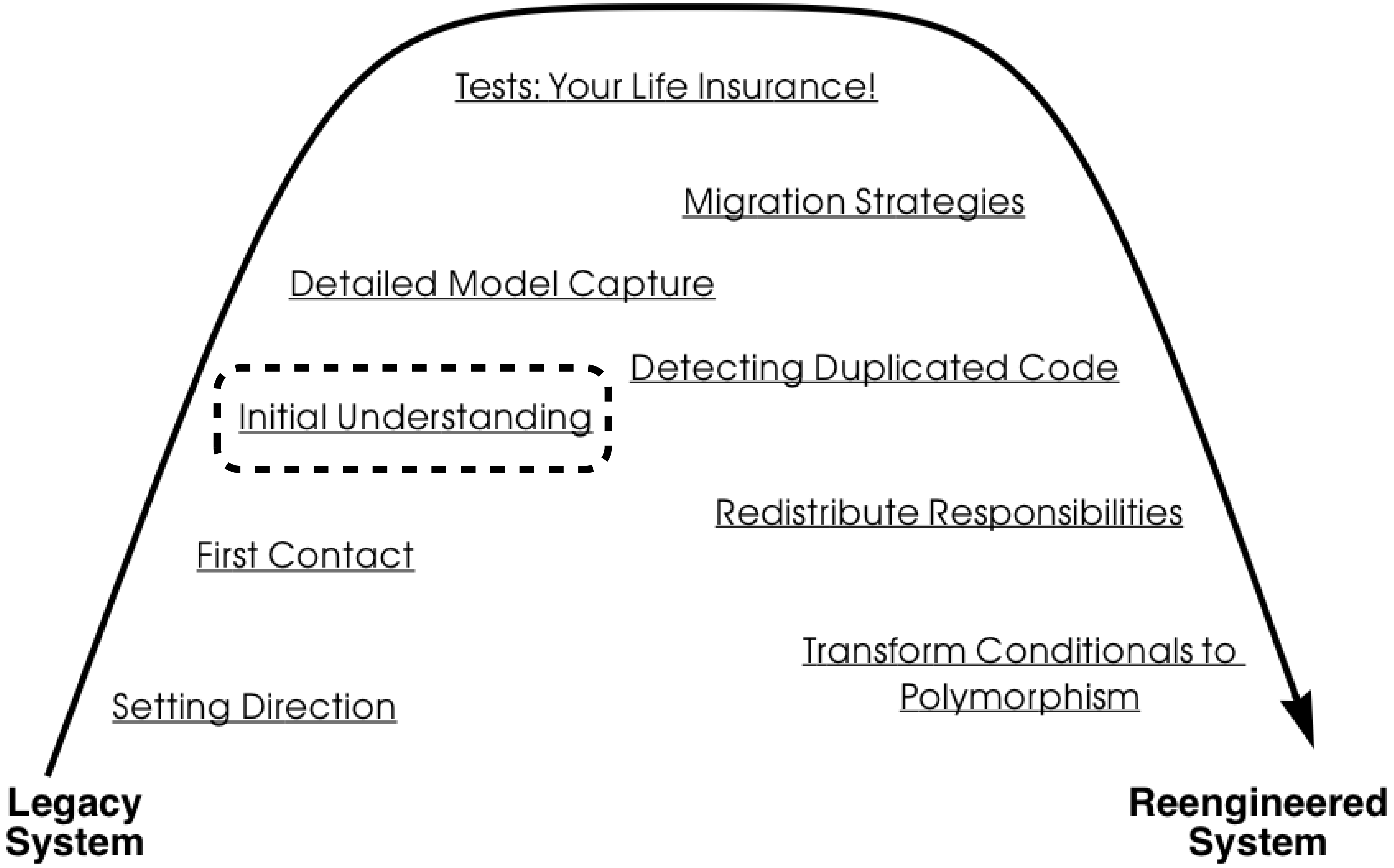
- Chat with the developers/maintainers
- Skim the documentation
- Read the code in one hour
  - A brief assessment of the source code in an intensive review
  - Focus on unit tests, write down question you intend to answer
  - Identify high level modules, e.g., UI, network layers, etc.
- If possible play with a running version of the system

Interview During the Demo

- Let users show you the functionality of a system
- It will give you some usage scenarios
- The main features of the system:
  - Which features are important?
  - Which features are less important?
  - What do user like/dislike about the system?
- Consider different stakeholders for an demo-interview

Chat with the Maintainers

- Which modules contain most of the defects?
- Which modules are changed often?
- How long does it take for a newcomer to understand the system?
- Which was the easiest/most difficult defect to fix?
- Why was it easy/difficult?
- How are priorities given? (Find out if there is a structured development process)



Initial Understanding (Ch. 4)

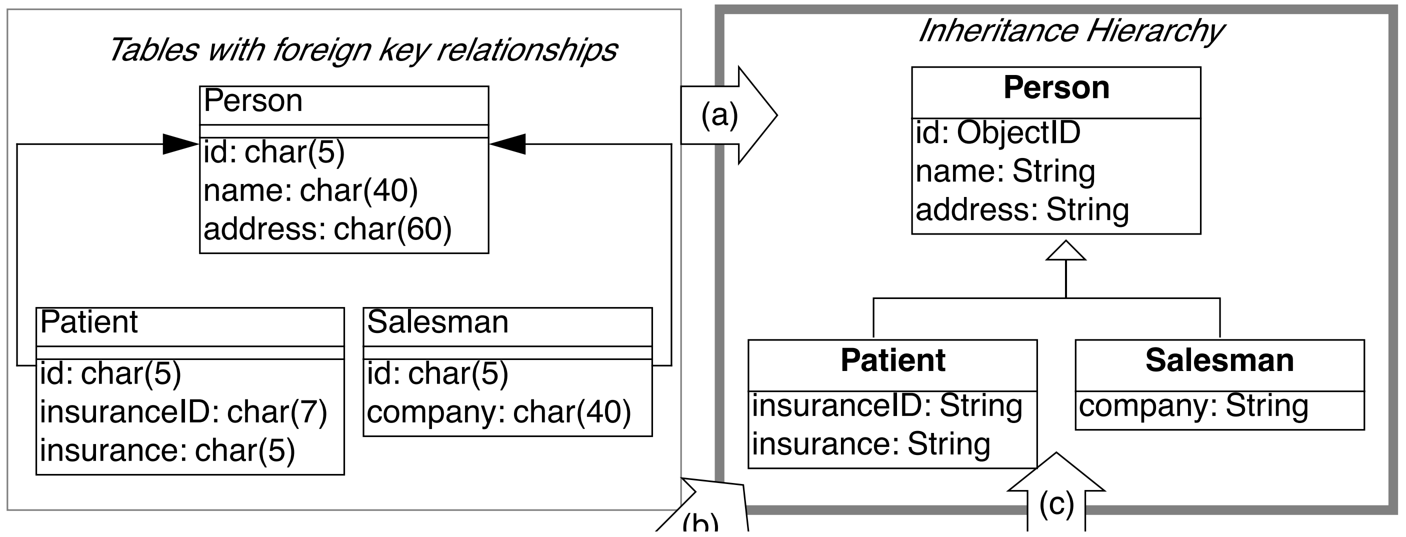
Get a deeper understanding

Initial Understanding (Chapter 4)

- Refine the ideas from the first contact into a deeper, initial understanding
- Layout the *foundation* for the rest of the project
- Large reengineering *projects take time*, therefore, *document*, in particular, initial decisions and their rationals
- This most likely is an *iterative* process

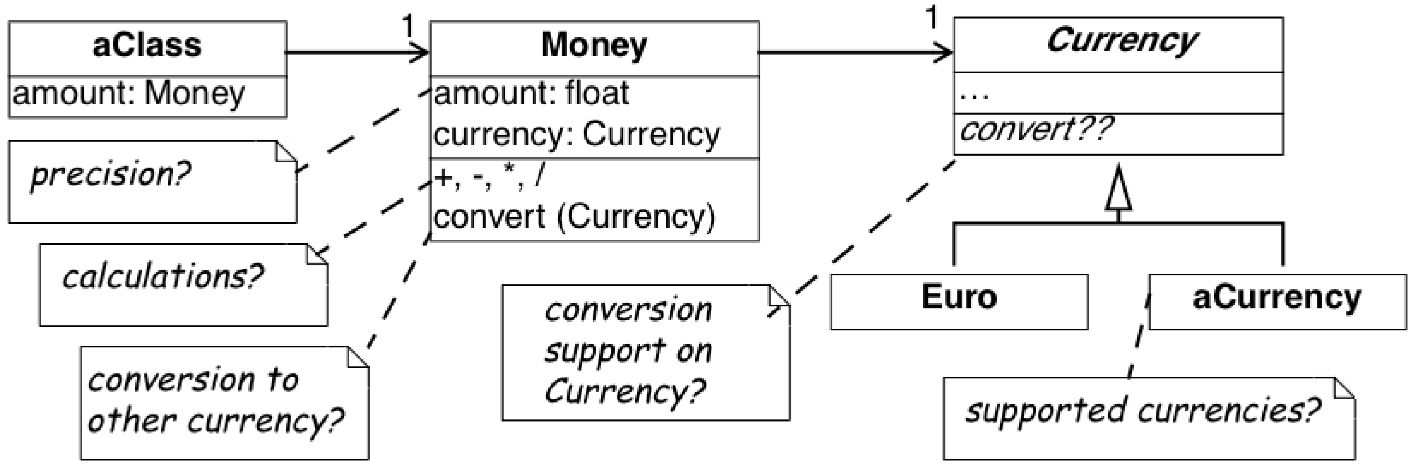
Patterns

- Analyze the persistent data
  - Data is the most precious asset
  - Look at database scheme and try mapping it to code entities
  - Foreign-Keys -> Class associations
  - Table -> Class
  - Columns -> Attributes
  - Class model should not be data driven



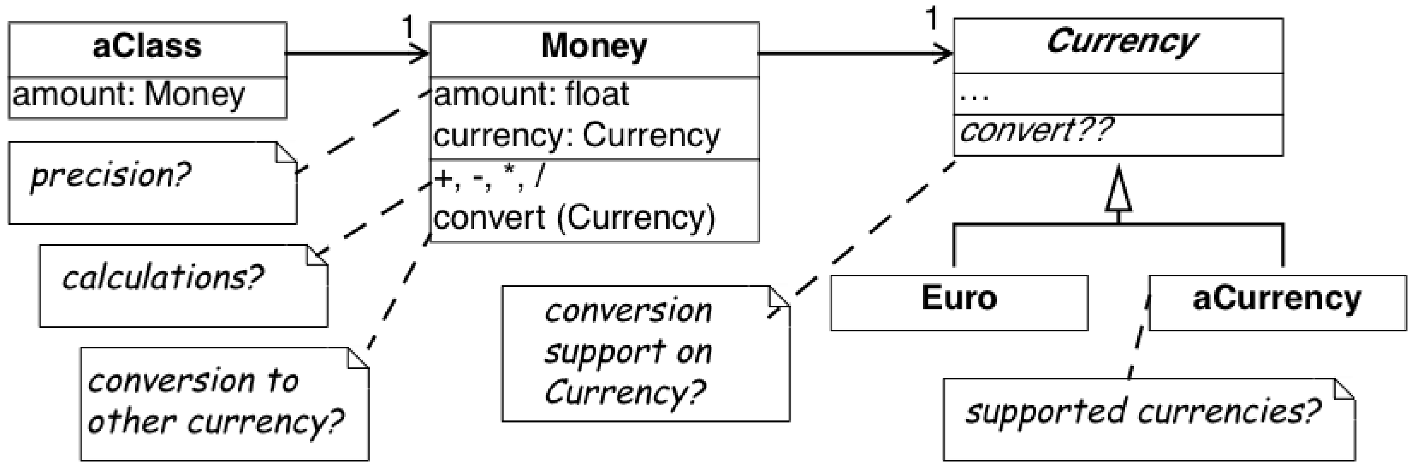
Patterns

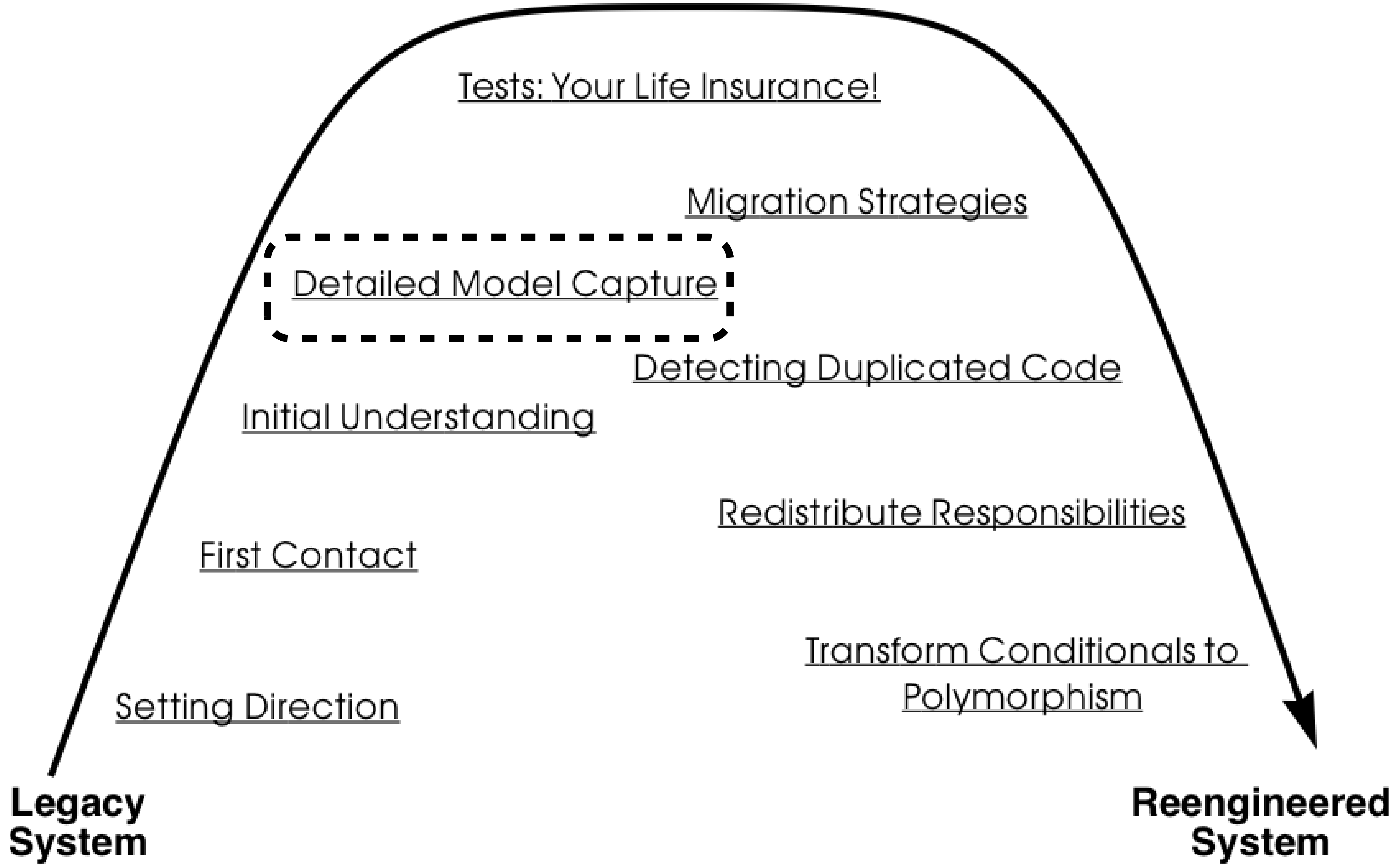
- Speculate about the design
  - Progressively recover design from code code
  - Draw plausible diagrams
  - Speculate about business objects:
  - How is the domain represented as classes?
  - Speculate about patterns:
  - Look for Design Patterns



Patterns

- What does it mean if we find an Observer Pattern?
- Maybe there is a kind of Publish-Subscriber design
- What does the occurrence of a Concurrency Pattern mean?
- ...





Detailed Model Capture (Ch. 5)

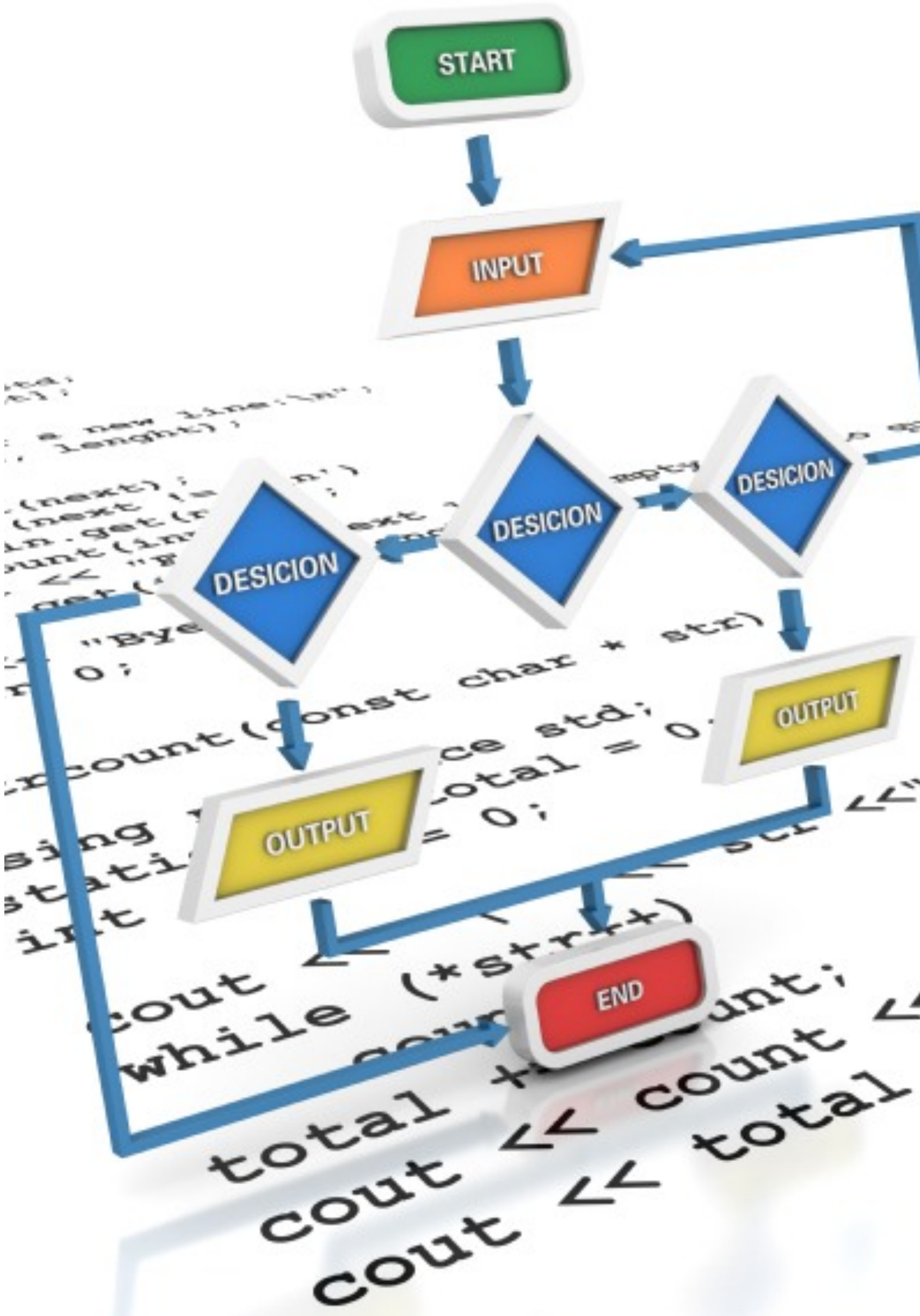
Understand the code, statically and dynamically

Detailed Model Capture (Chapter 5)

- The important, most valuable parts are identified, and we have a decent understanding of those parts
- Build a deep and detailed understanding on fine-grained code level
- Expose design artifacts hidden in the code
- More technical than the previous steps
- Dynamic vs. static structure

Step Through Execution

- Static Structure
  - Polymorphism: Which concrete object are instantiated at runtime?
  - When are they instantiated?
  - Code reveals the dependencies of a system, but not a the actual collaboration
  - Which sequence of events do occur at runtime? What is the actual flow of execution?
- Debug typical scenarios and enhance the current understanding of the code with this knowledge of the runtime behavior
  - What is an appropriate scenario for debugging?



Learn from the Past 1

- Legacy system evolved over a long time through many iterations and changes
- The current design of a system the result of a *continuous process* triggered by a multitude of external and internal factors
- Development history is an extremely valuable source for information
- Compare subsequents revisions of the code
- Problem: 1000 source files in the current version; each file

  has on avg. 20 revisions: Does not scale for large projects

Learn from the Past 2

- Focus on those parts that change the most often: Sign of potential *unstable design*
- Try to *understand* why those parts change so that often
- Unstable design offer great opportunities for reengineering
- *Stable parts* should not be the primary focus (especially if time is scarce)
- Use appropriate tools to identify the unstable parts
- Entire lecture will be dedicated to *software visualization*
- Lecture on *empirical software engineering*

Refactor to Understand 1

- Cryptic code is hard to grasp and understand
- Such code hardly reflects its purpose
    - meaninglessly or confusingly named attributes, method names, or parameters

```
const long ZHANYIZHOUQI = 1000; //1000
 const long ZHANYIZHOUQI_WITH_NOZZLE_CHANGES = 100; //
 const int ENDING_SLOT_NUMBER_OF_FRONT_SIDE = 38;
 const int NOT_SHIELD = 0;
 const double PANZIGAILUU = 0.1;
```

```
const int TWENTY_EIGHT = 28;
```

```
public class Student {

     private String myString;

     public Student(String theString) {
          myString = theString;

          // ...
     }

  }
```

```
/* important global variables */
unsigned int x;
unsigned int xx;
unsigned int y;
unsigned int yy;
```

http://c2.com/cgi/wiki?BadVariableNames

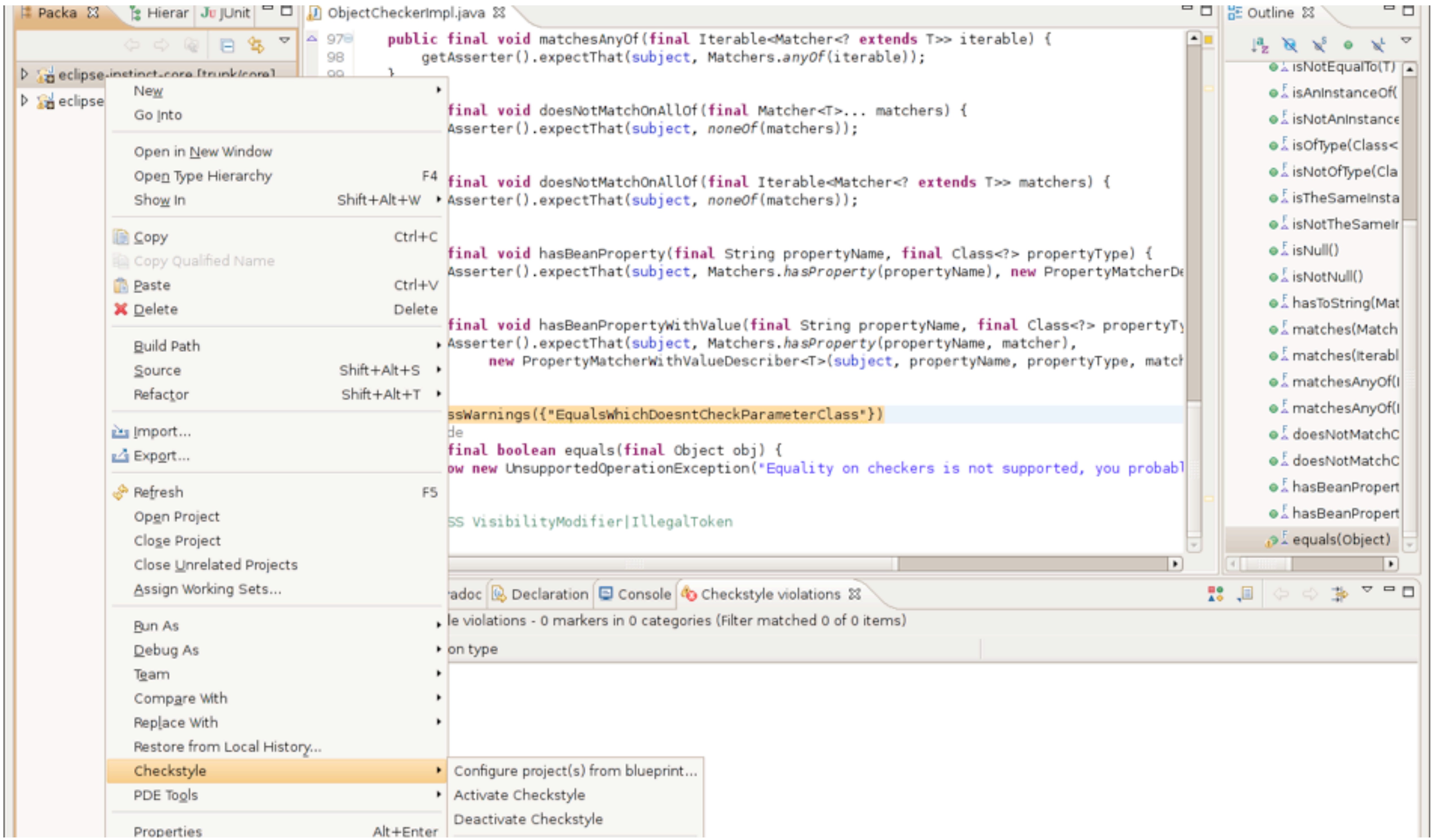
```
public class Foo {
     public void setPassword(String password) {
          // don't do this
          if (password.length() > 7) {
               throw new InvalidArgumentException("password");
          }  }
```

```
public class Foo {
     public static final int MAX_PASSWORD_SIZE = 7;

     public void setPassword(String password) {
          if (password.length() > MAX_PASSWORD_SIZE) {
               throw new InvalidArgumentException("password");
          }
}
```

Magic Numbers

Plain Numbers in Code



```
    public final void matchesAnyOf(final Iterable<Matcher<? extends T>> iterable) {
        getAsserter().expectThat(subject, Matchers.anyOf(iterable));
    }

final void doesNotMatchOnAllOf(final Matcher<T>... matchers) {
Asserter().expectThat(subject, noneOf(matchers));

final void doesNotMatchOnAllOf(final Iterable<Matcher<? extends T>> matchers) {
Asserter().expectThat(subject, noneOf(matchers));

final void hasBeanProperty(final String propertyName, final Class<?> propertyType) {
Asserter().expectThat(subject, Matchers.hasProperty(propertyName), new PropertyMatcherDe

final void hasBeanPropertyWithValue(final String propertyName, final Class<?> propertyTy
Asserter().expectThat(subject, Matchers.hasProperty(propertyName, matcher),
    new PropertyMatcherWithValueDescriber<T>(subject, propertyName, propertyType, match

ssWarnings({"EqualsWhichDoesntCheckParameterClass"})
de
final boolean equals(final Object obj) {
ow new UnsupportedOperationException("Equality on checkers is not supported, you probabl

SS VisibilityModifier|IllegalToken
```

le violations - 0 markers in 0 categories (Filter matched 0 of 0 items)

Check Style

http://checkstyle.sourceforge.net/availablechecks.html

Refactor to Understand 1

- Cryptic code is hard to grasp and understand
- Such code hardly reflects its own purpose
  - meaninglessly or confusingly named attributes, method names, or parameters
  - The order of the statements may not matter for the execution, but they may matter when reading the code. For instance, handle default control flow in the if-part
- Refactor the code making it reflect its purpose
- *Primary* goal is to *understand*, not to reengineer (“*refactoring experiments*”)
- Refactoring means changing code: *Unit tests are essential*!

Refactor to Understand 2

- Refactoring *does not change the functionality* of a system
- Transforming the system into a better shape with respect to Software Engineering principles.
- In contrast: After reengineering that system can have new features

Look for Contracts

- What does a class expect of its clients?
- A.k.a. How do I use an API?
    - Proper sequence of method calls
    - How do I instantiate a certain object of a class?
- Documentation is out of synch
- Look for certain, similar code block related to an API
- Reason about contracts, document contracts, run unit tests

At this point we have enlightened ourselves with a detailed understanding of the system and the crucial parts to be reengineered

We are now ready to reengineer

Document everything and *record all the knowledge* of the reverse engineering phase (and continue doing so!)

Lecture Outlook

- Reengineering
- Software Analysis Visualization
- Modeling the history of a project
- Empirical SWE