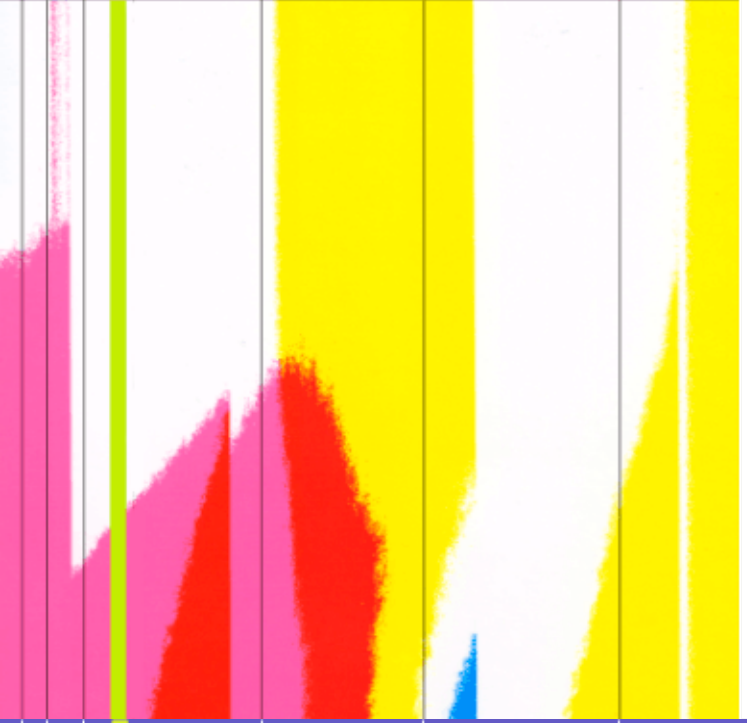
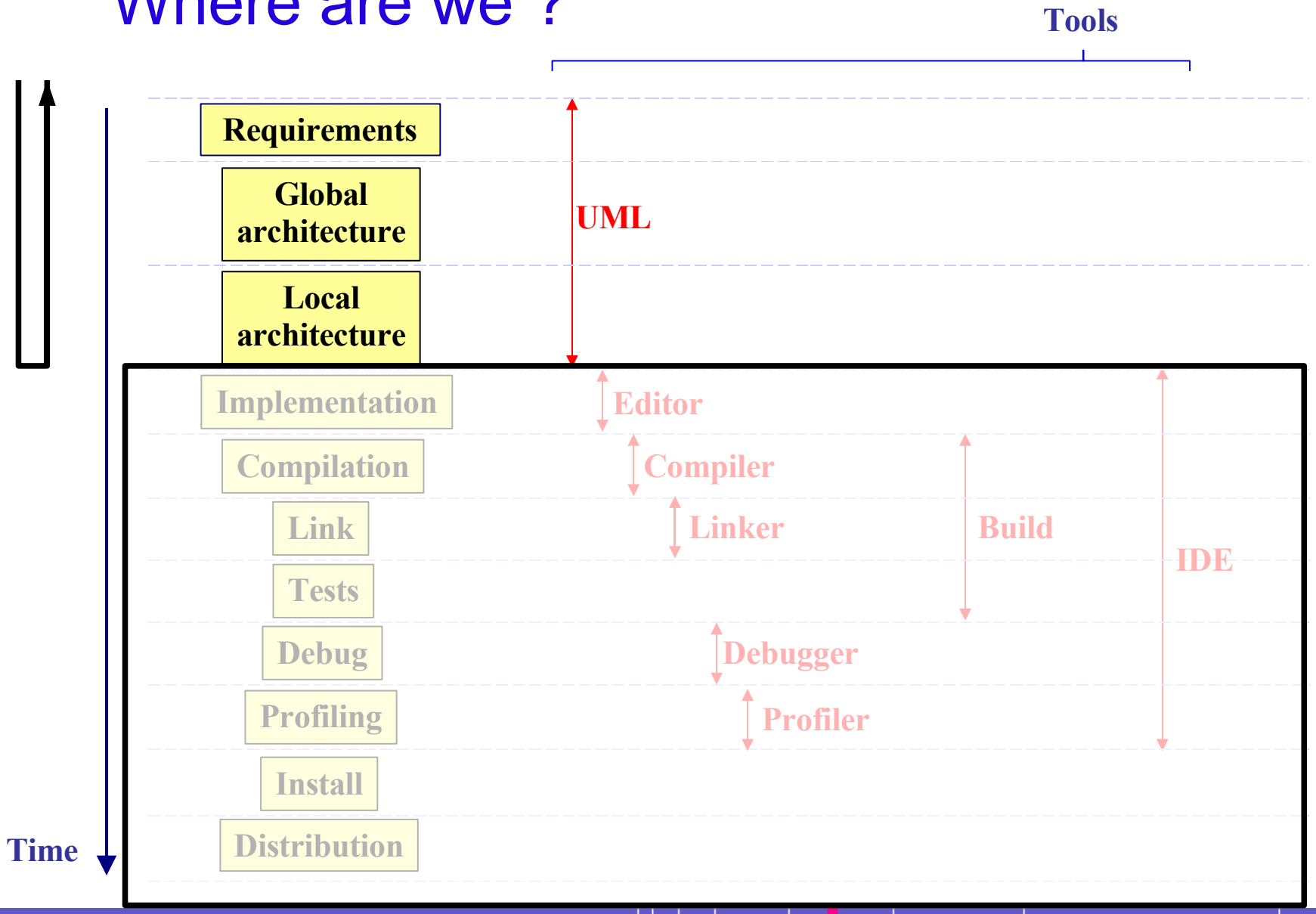


Requirements and Design

Erwan Demairy - Dream

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

Where are we ?


Tools
Requirements
Global architecture
Local architecture
UML
Implementation
Compilation
Link
Tests
Debug
Profiling
Install
Distribution
Editor
Compiler
Linker
Build
IDE
Debugger
Profiler
Time

Analysing requirements and designing

Requirements analysis

- Translate the needs from user-language to computer scientist language
  - Only time, communication, empathy and patience can help

Difficulties when designing

- Old code tend to be cluttered => refactoring
- It is hard to structure well from scratch => class discovery, design patterns
  - Read as much code as you can
  - Read your own code some time latter
    - Ability growth with practice
  - Read at least the books of the Dream list (1 book/month)

Overview

Introduction

What should be produced ?

Design:the bottom-up approach

Design:the top-down approach

Conclusion

Why “waste” time writing documents?

Understand the software without reading code

Help with development planning by foreseeing the risks

Team development

- Collaborative work
- Implementation continuity with several developpers

Requirements and Design: Documentation

Requirements: define the development's limits

- Requirements specification: 2-3 pages at most
- Prototypes for specific risks

Design: architecture of the software

- Diagrams (e.g. UML): use cases, classes, sequences, ...
- Text description about the software design (<10 pages)
- Prototypes when new risks discovered

To be revised when necessary (even during implementation)

Must be archived to keep trace of revisions (e.g., with SVN)

Examples of outlines: http://readyset.tigris.org/

Requirements Document Outline: Example

1. Scope
   1. Overview
   2. Logic Requirements
   3. Required Algorithms
   4. Example of the Software Usage
   5. Future Component Direction
2. Interface Requirements
   1. Provided interfaces
      1. Graphical User Interface (GUI)
      2. Program interface (API)
   2. Required interfaces
      1. 3rd party libraries (Posix, MPI, ...)
      2. Platform (X11, win32, ...)
   3. Namespace
3. Software Requirements
   1. Administration Requirements
   2. Technical Constraints
      1. Software Component Dependencies:
      2. QA Environment:
      3. Performance
   3. Design Constraints

Matrix Math Library 2.0 Requirements Specification

1. Scope

1.1 Overview

The Matrix Math Library component is a Java library for performing operations on matrices. This includes basics like addition and multiplication, as well as more complex operations like computing determinants and eigenvectors.

Singular Value Decomposition is introduced with the 2.0 version of this component. This version includes updates to the existing methods as well as all new methods required to support SVD.

1.3 Required Algorithms

Articulate your choice of SVD algorithm, or explain the procedure you develop. *ENTRIES LACKING DETAILED EXPLANATIONS OF THE SVD ALGORITHM WILL BE REJECTED AND **NOT** REVIEWED.*

2. Interface Requirements

2.1.1 Graphical User Interface Requirements

None

2.1.2 External Interfaces

None specified

2.1.3 Environment Requirements

- Development language: J2SE 1.4
- Compile target: J2SE 1.2, J2SE 1.3, J2SE 1.4, J2SE 1.5

Text description of the software design: outline example

1. Description of the internals of the software
    1. Architecture of the classes, eg used design patterns
    2. Purpose of each class
    3. Detailed algorithm (i.e. pseudo-code)
2. Runtime environment requirements:
    OS and software (JVM version...), 3rd party libraries, ...
3. Installation and configuration: i.e. *README.txt*, *INSTALL.txt*
4. Usage
    - How to run automated tests
    - Examples of the software's use

Overview

Introduction

What should be produced ?

Design bottom-up

Design Top-down

Conclusion

Design bottom-up: refactoring

From existing code

- Quick and dirty implementation
- Pre-existing implementation

Change only the structure of the code: the behavior stays the same

Only small changes

→ Resulting code better structured

Mandatory tools

- Version management system (CVS, SVN...)
- Regression tests

How to do a small change

1. Find a “smelling” part
2. Apply the receipt
    1. Write/rewrite tests (to check behavior stability)
    2. Refactor the small part
    3. Compile + test it to check the behavior stability
3. Iterate

Receipts book: Refactoring textbook

Refactoring – Martin Fowler

Most common smelling structures to refactor

1. Duplicated Code
   - Example:copy-paste of a method to handle a related case
2. Long Method
   - Example:methods of thousands lines
3. Large Class
   - Example:Class having several purpose
4. Long Parameter List
   - Divergent Change

Risks

- Reduce readability
- Impair maintainability of the code

Example of refactorisation:extract method

```
Void printOwing(double amount) {
    • PrintBanner();
    • // print details
    • System.out.println("name:"+_name);
    • System.out.println("amount"+amount);
}
```

Long method

```
Void printOwing(double amount) {
    PrintBanner();
    printDetails(amount);
}
void printDetails(double amount) {
    System.out.println("name:"+_name);
    System.out.println("amount"+amount);
}
```

Shorter methods

Overview

Introduction

What should be produced ?

Design bottom-up

Design Top-down

Conclusion

Design Top-down

From scratch

Find the classes

- What are the relevant classes for your software?
- What are their responsabilities?
- How do they interact each other?

How to discover the classes?

"Natural" classes

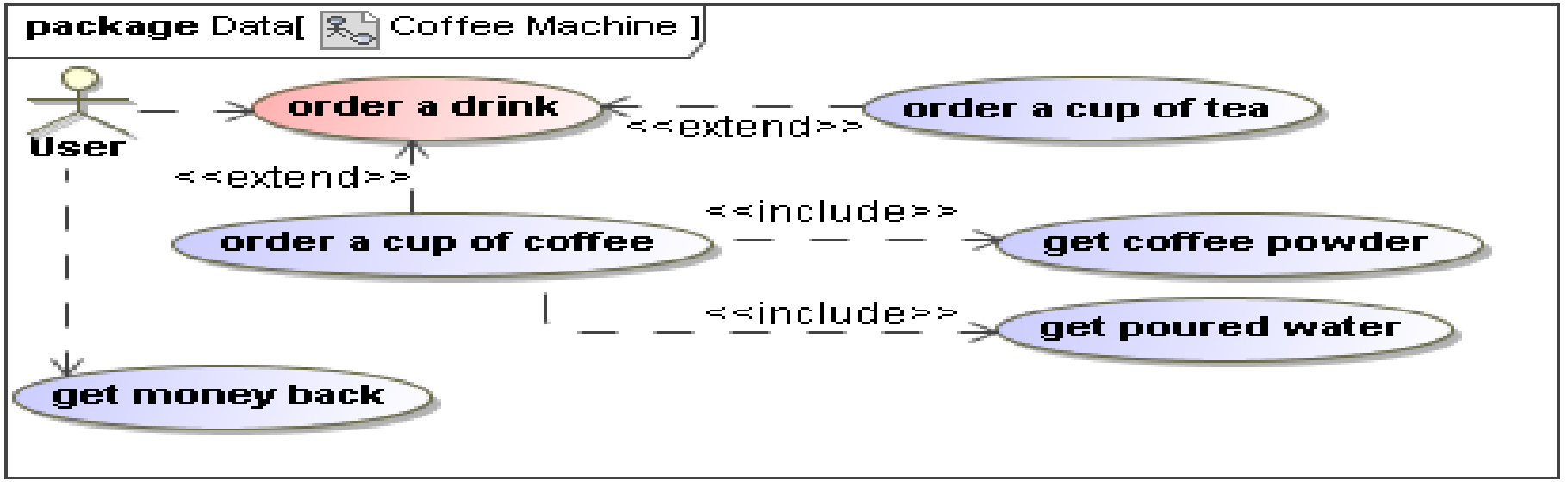
- Example:Airplane, Reservation

Verb + Noun method / Use case based

- Noun = candidate classes, Verb = method of the class

CRC

Verb+noun/use case:coffee machine example



Order a drink => Class Drink with an order method

Help to find main relevant classes

Limitations

- Responsabilities: does not prevent cluttered classes
- Interactions between classes are missing

Good starting point

Class-Responsibility-Collaboration (CRC) cards

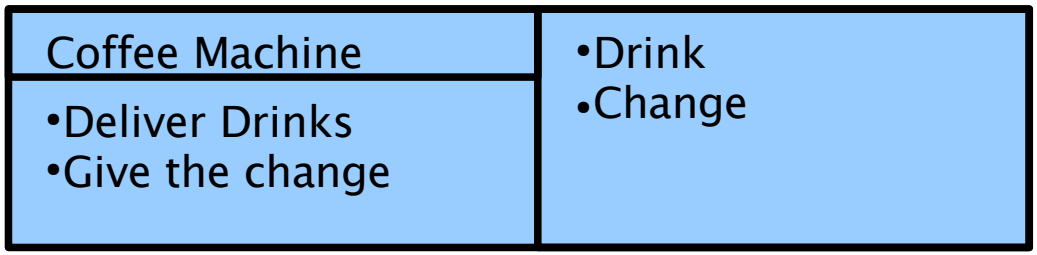
One (small, i.e. A5) card = one candidate class

- Divide the card in three parts
  - The class name
  - Responsibilities of the class.
  - Names of other classes that the class will collaborate with to fulfill its responsibilities.
- You can add
  - Author
  - Super and Sub classes (if applicable)

Better role-played with 2 or more developers

Class Name	Collaborations
Responsabilities	

CRC Example:the coffee machine



CRC Example:the coffee machine

Coffee Machine	•Drink •Change
•Deliver Drinks •Give the change •Bookkeeping of available parts	

Drink	•Coffee Machine
•Mix the parts	

CRC Example:the coffee machine

Coffee Machine	•Drink •Change
•Deliver Drinks •Give the change •Bookkeeping of available parts	

Drink	•Coffee Machine
•Mix the parts	

Change	
•Compute the change •Bookkeeping of the available coins	

Design Patterns

Well-known, efficient structures

- It helps to document the code, since they are well-known
- Less development time
- Fewer mistakes

Basic building blocks divided in three main categories

- Creational: for class instanciation
- Structural: class and object composition
- Behavioral: objects communications

See the following for a gentle introduction

- Design Patterns Explained - A. Shalloway, J.R. Trott

Overview

Introduction

What should be produced ?

Design bottom-up

Design Top-down

Conclusion

Conclusion

1. Document your requirements and design!

2. When designing : two complementary approaches
    1. Software refactoring
        1. From existing code
        2. Iterative programming
    2. Software design
        1. From scratch
        2. Verb+noun / CRC / Design patterns

3. Read, discuss and practice software engineering!

References

Page dream: http://www-sop.inria.fr/dream/

Modélisation objet avec UML – Muller, Gaertner

UML Distilled – Martin Fowler

Refactoring – Martin Fowler

Design Patterns Explained - A. Shalloway, J.R. Trott