Software Evolution

Lecture 5: Clone Detection and Management

Status check

- Annotated Bibliography
  - Include reading of week 1 through 5 (at least)
    Deadline is week 8.
  - Are you up-to-date?
    Read every week!
  - Do you have sufficient feedback?
    Write, review and refine. Ask for feedback!

- Practical Lab Series 2: Clone Detection

This week's reading

- Rainer Koschke. Software Evolution, chapter 2. Identifying and Removing Software Clones, pages 15–36. Springer, 2008.

- C. Kapser, M.W. Godfrey, *"Cloning Considered Harmful" Considered Harmful*, in Proceedings of the 13th Working Conference on Reverse Engineering, 2006.

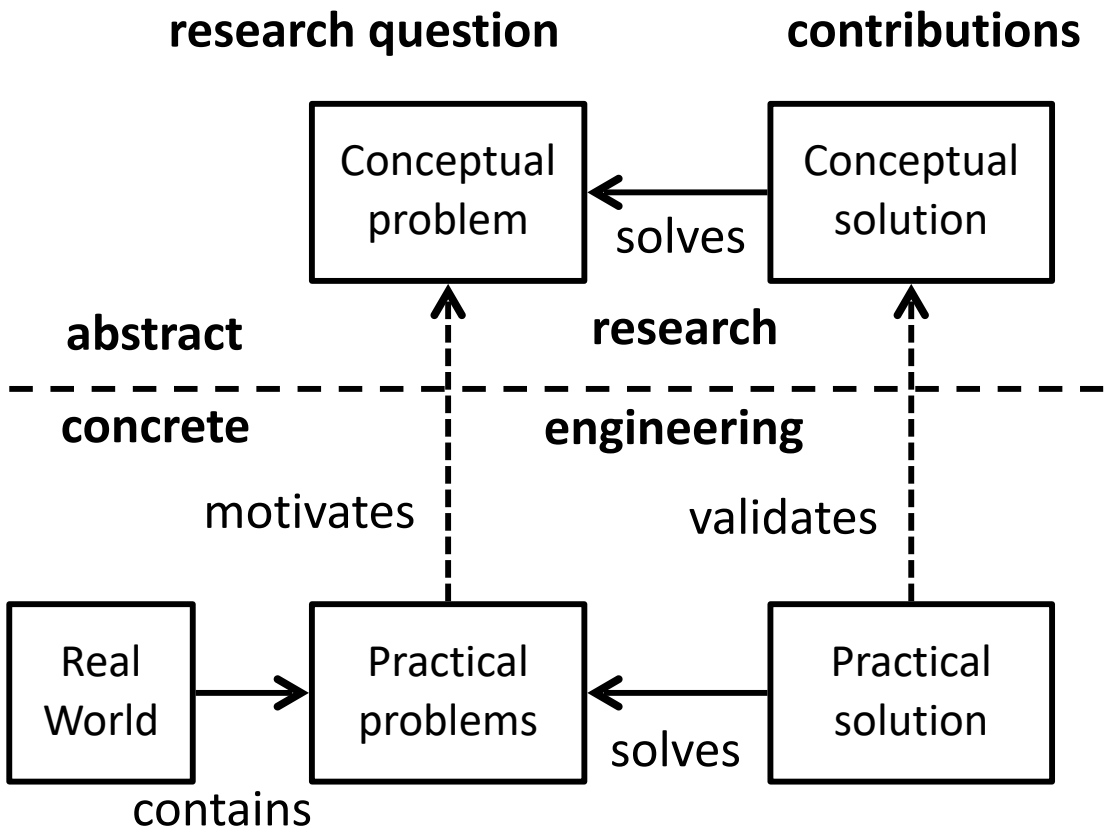
Code duplication, cloning, redundancy

- Important aspect of evolving software
- Active research field with lots of results
- Interesting opportunity for constructing tools

- **Relevance.** Cloning happens in practice
  - 7 – 23 % duplication in general, can be as high as 50%

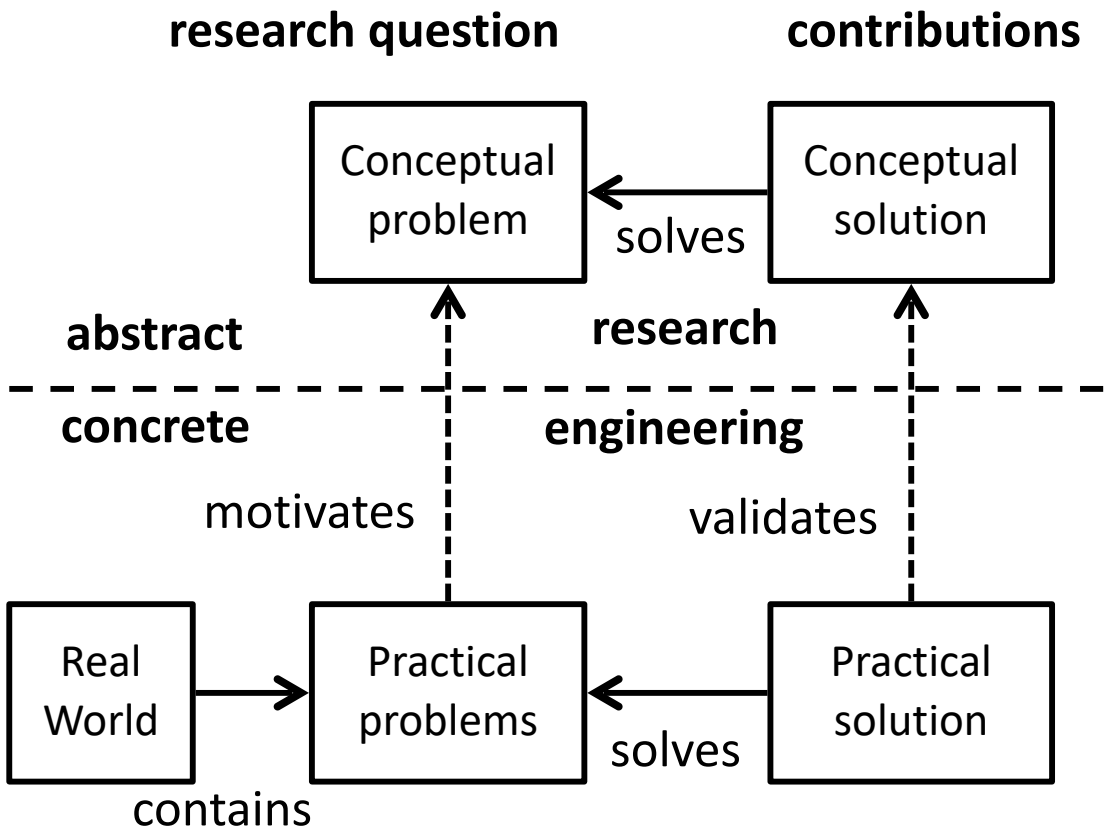
Frameworks for reasoning, understanding and communicating

- **Problem:** Defines an urgent relevant problem that motivates research
- **Objectives:** Sets a well-scoped goal with respect to the problem. Poses a conceptual *research question*.
- **Approach:** outlines practical steps towards attaining objectives. Answers research question. Enumerates *contributions* (claims)



Code duplication, cloning, redundancy

- Motivation
  - Improve code quality
  - Avoid redundant code
  - Improve reuse
  - Reduce size
  - Reduce maintenance effort
  - Expose plagiarism
  - Expose license issues
  - Gain insight into how code evolves and how programmers work: e.g, forking, copy-pasting, templating, etc.
  - “manage” clones



What this course will teach you

- The basic concepts and “laws” of software evolution
- How to measure or detect evolution in real software (i.e., code)
- How to build tools to automate dealing with existing software
- Scientific reading and reporting

The plan for today

1. Clone detection
   - Finding clones
   - Visualizing clones
2. Break (15 minutes max.)
3. Clone management
   - Preventive
   - Compensative
   - Corrective
4. Lab series 2

Cloning definitions

Clone: fragment of code that is *duplicated* somewhere else.

Clone pair: two code fragments that are duplicates of each other.

Clone class: any number of code fragments that are all duplicates of each other.

More formally

Consider ‘is clone of’ an *equivalence relation* <->

Reflexivity:

every clone is a clone of itself

Symmetric:

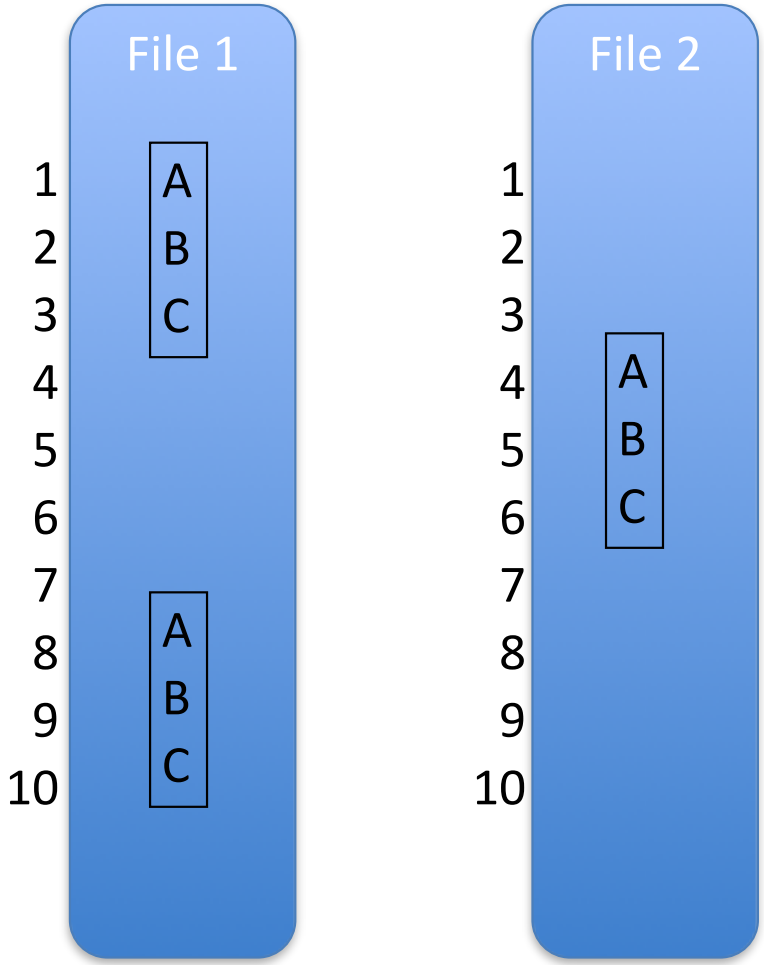
if A <-> B, then B <-> A

Transitivity:

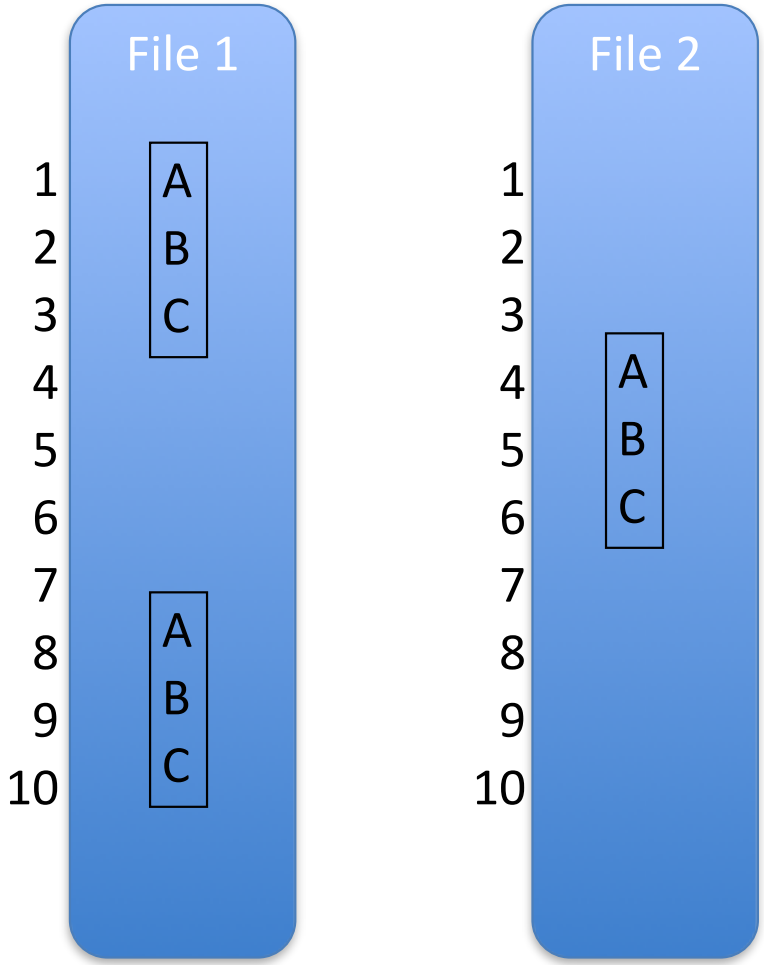
if A <-> B and B <-> C, then A <-> C

Given the above, clone classes are the *equivalence classes* of <->

Cloning definitions



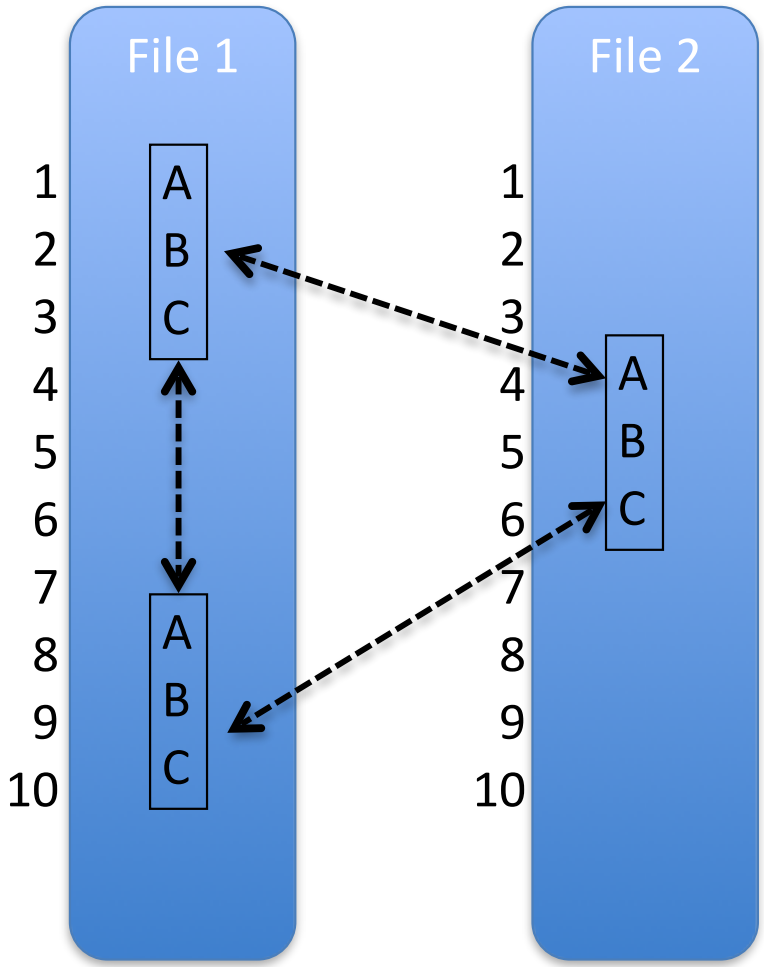
Cloning definitions: clones



Clones:

1. File 1: 1-3
2. File 1: 8-10
3. File 2: 4-6

Cloning definitions: clone pairs



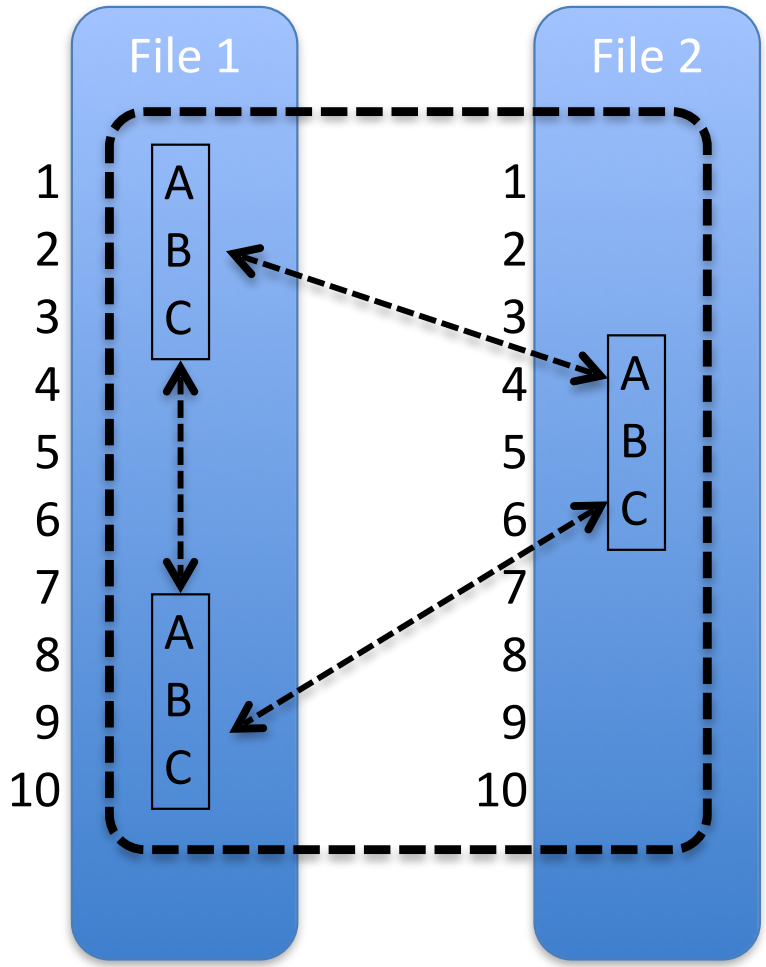
Clones:

1. File 1: 1-3
2. File 1: 8-10
3. File 2: 4-6

Clone pairs:

1. File 1: 1-3 <-> File 1: 8-10
2. File 1: 1-3 <-> File 2: 4-6
3. File 1: 8-10 <-> File 2: 4-6

Cloning definitions: clone classes



Clones:

1. File 1: 1-3
2. File 1: 8-10
3. File 2: 4-6

Clone pairs:

1. File 1: 1-3 <-> File 1: 8-10
2. File 1: 1-3 <-> File 2: 4-6
3. File 1: 8-10 <-> File 2: 4-6

Clone classes:

1. File 1: 1-3, File 1: 8-10, File 2: 4-6

Clone types

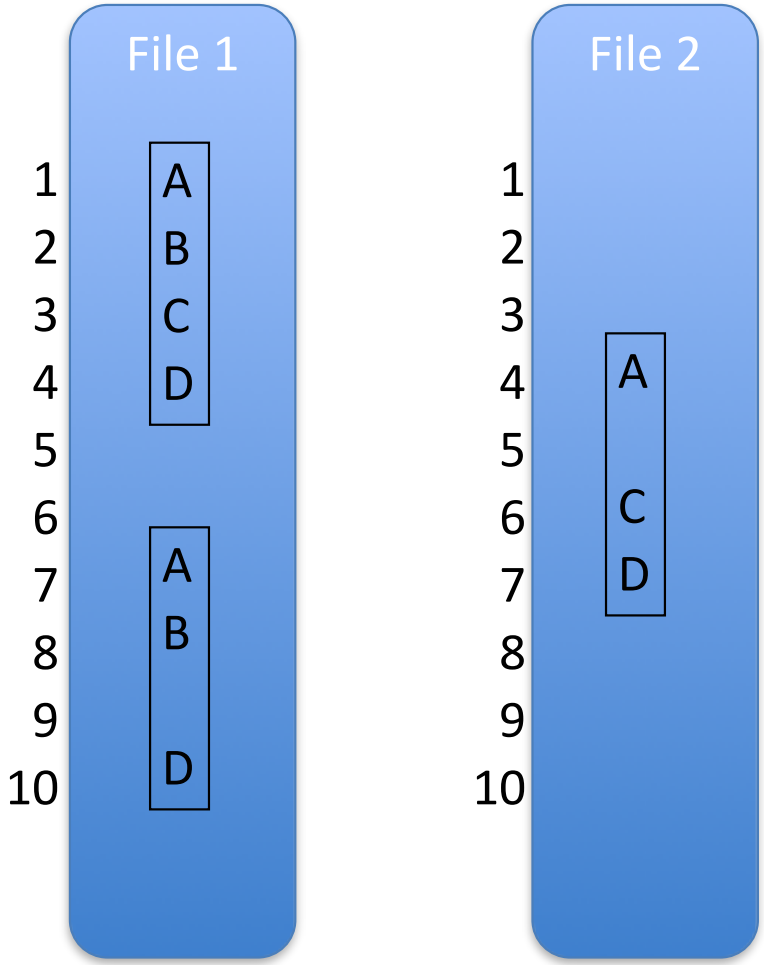
Type 1: exact copy, ignoring whitespace and comments.

Type 2: syntactical copy, changes allowed in variable, type, function identifiers.

Type 3: copy with changed, added, and deleted statements.

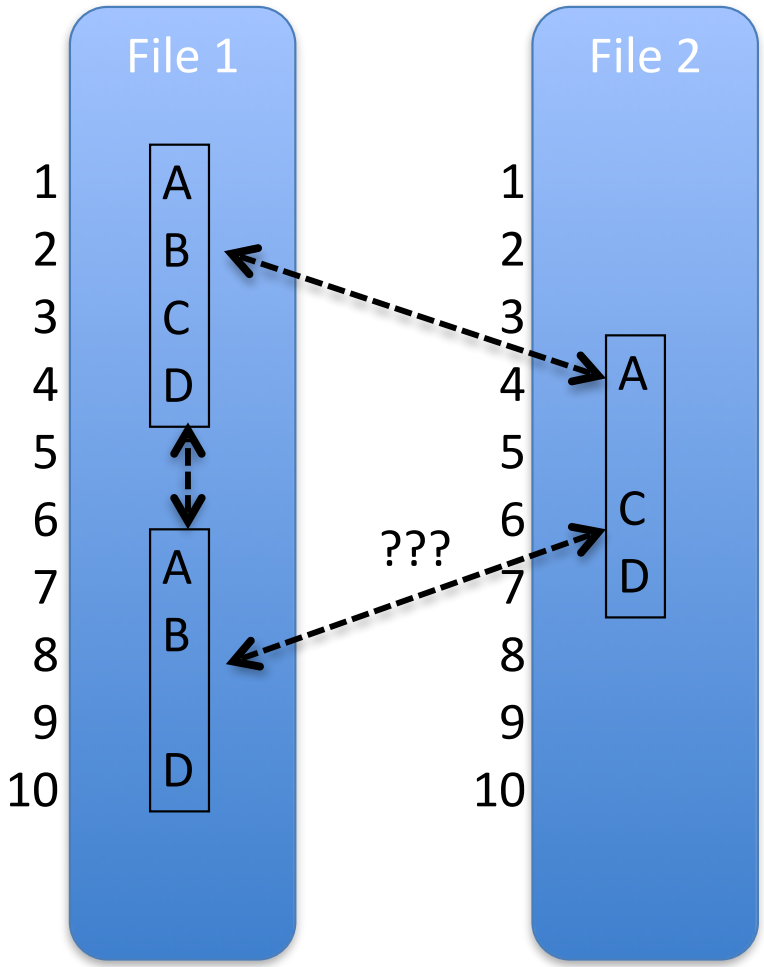
Type 4: functionality is the same, code may be completely different.

Cloning definitions: Type 3?



Example: clones can have 1 line added/deleted.

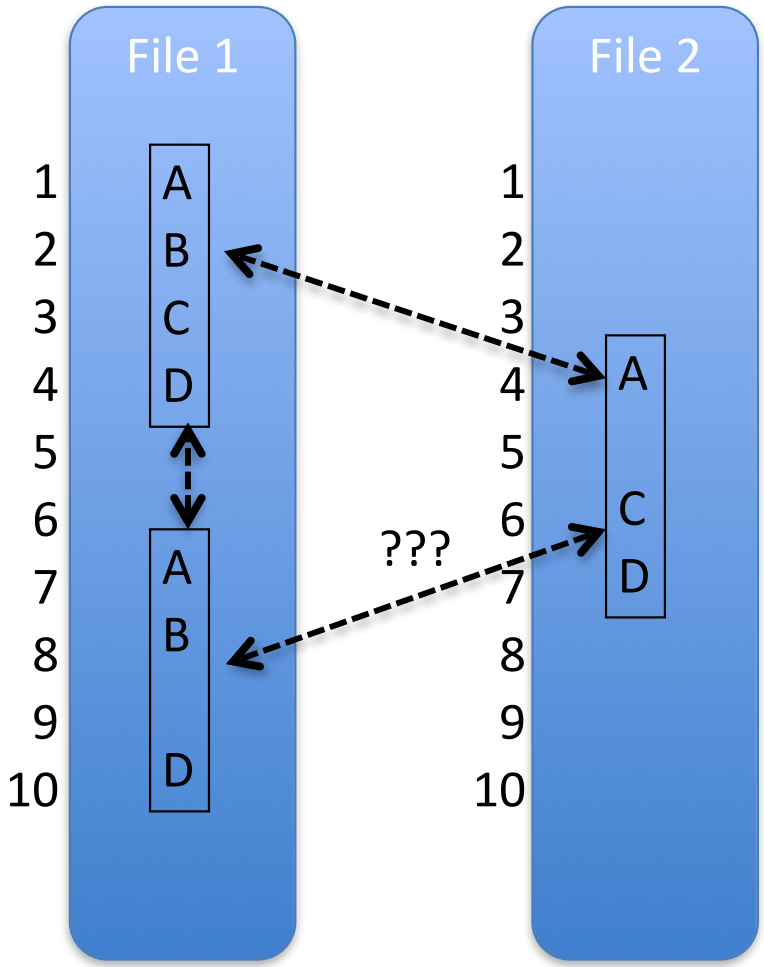
Cloning definitions: Type 3?



Clone pairs

1. File 1: 1-4 <-> File 1: 7-8, 10
2. File 1: 1-4 <-> File 2: 4, 5-7
3. File 1: 7-8, 10 <-> File 2: 4, 5-7 ???

Cloning definitions: Type 3?



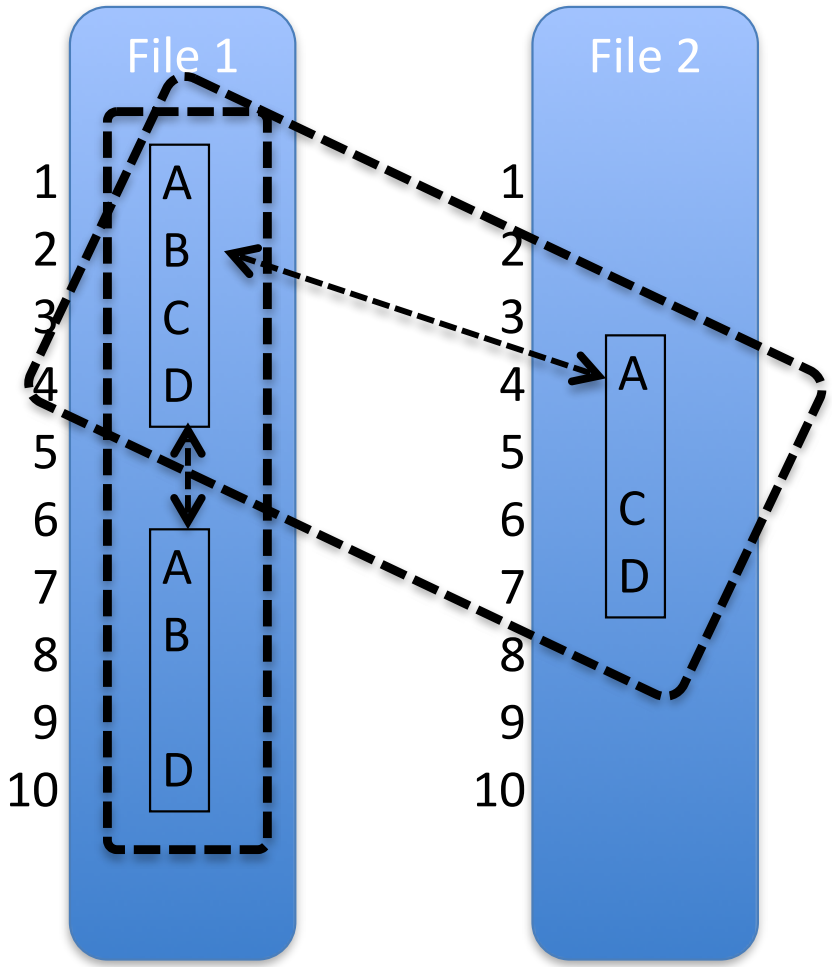
Clone pairs

1. File 1: 1-4 <-> File 1: 7-8, 10
2. File 1: 1-4 <-> File 2: 4, 5-7
3. File 1: 7-8, 10 <-> File 2: 4, 5-7 ???

‘is Type 3 clone of’ relation is not transitive!

What are the clone classes here?

Cloning definitions: Type 3?



Clone pairs

1. File 1: 1-4 <-> File 1: 7-8, 10
2. File 1: 1-4 <-> File 2: 4, 5-7
3. File 1: 7-8, 10 <-> File 2: 4, 5-7 ???

'is Type 3 clone of' relation is not transitive!

What are the clone classes here?

1. File 1: 1-4, File 1: 7-8, 10
2. File 1: 1-4, File 2: 4, 6-7

Only Type 1 and Type 2 can be considered equivalence relations!

Clone detection algorithms

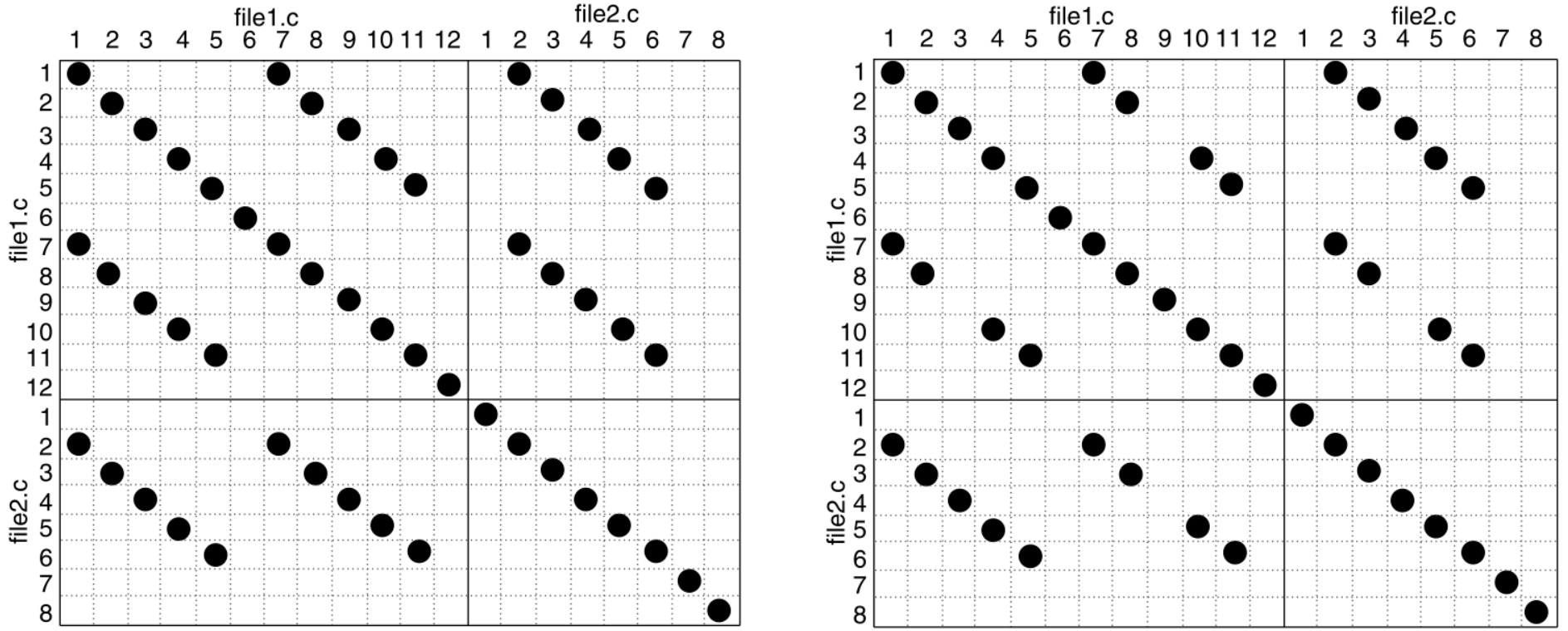
Algorithm can be based on:

- Text
- Tokens
- Metrics
- Abstract Syntax Trees (AST)
- Program Dependency Graphs (PDG)
- Other

Text-based clone detection

- **Compare lines.** Filter comments and compare each line to each other textually.
- Use hashing to speed up
- Extend cloned lines into bigger fragments

Clone visualization – dot plots



(a) Three clones

(b) Three clones, one modified line

Fig. 2.3. Dot plots [114, 156, 512]

Source: R. Koschke, *Identifying and Removing Software Clones,* in Software Evolution, Springer, 2008.

Token-based clone detection

- **Scan.** First apply lexical analysis to obtain tokens
- **Normalize.** Optionally apply normalization on the tokens
  - e.g. insert missing parentheses: if (a) … becomes if (a) {…}
  - Remove some tokens
- **Compare tokens.** Use an efficient algorithm (e.g. based on creating a suffix tree) to detect identical sequences of tokens

Example: Clone Detection Using Abstract Syntax Trees Clone Detection Using Abstract Syntax Suffix Trees = CDUASTCDUASS'T$

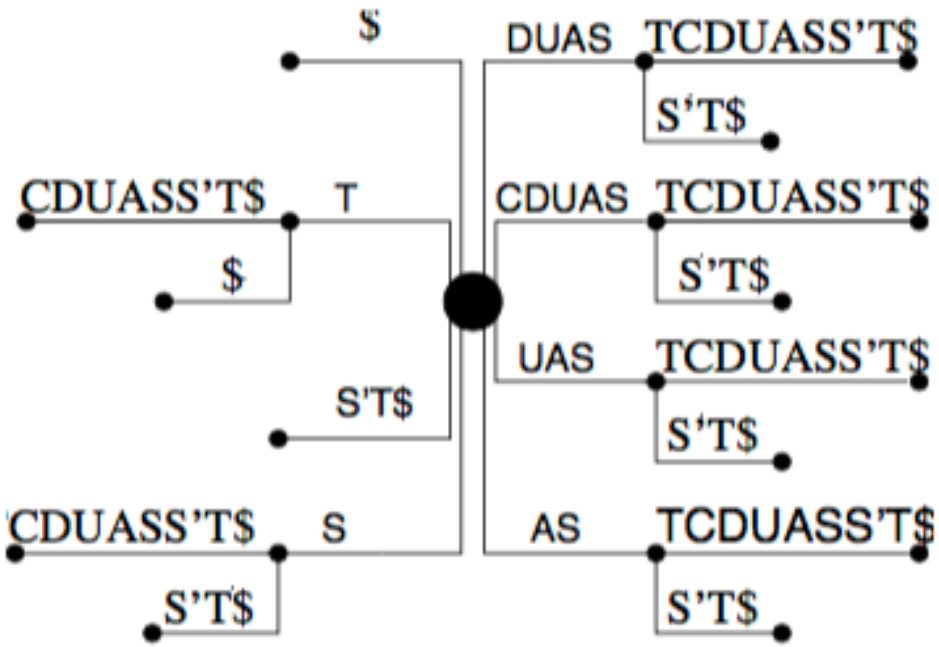


Figure 1. Suffix Tree for CDUASTCDUASS'T$; the large dot is the root

Koschke, Rainer, Raimar Falke, and Pierre Frenzel. "Clone detection using abstract syntax suffix trees." *2006 13th Working Conference on Reverse Engineering*. IEEE, 2006.

Metric-based clone detection

- **Collect code fragments.** Split the code into well-defined fragments, for instance methods or functions

- **Perform measurements.** Calculate vector of metrics on each fragment
  - Original approach uses 21 metrics on layout, control flow, and expressions.

- **Measure distance.** Calculate distance measure between vectors to find similar fragments

AST-based clone detection

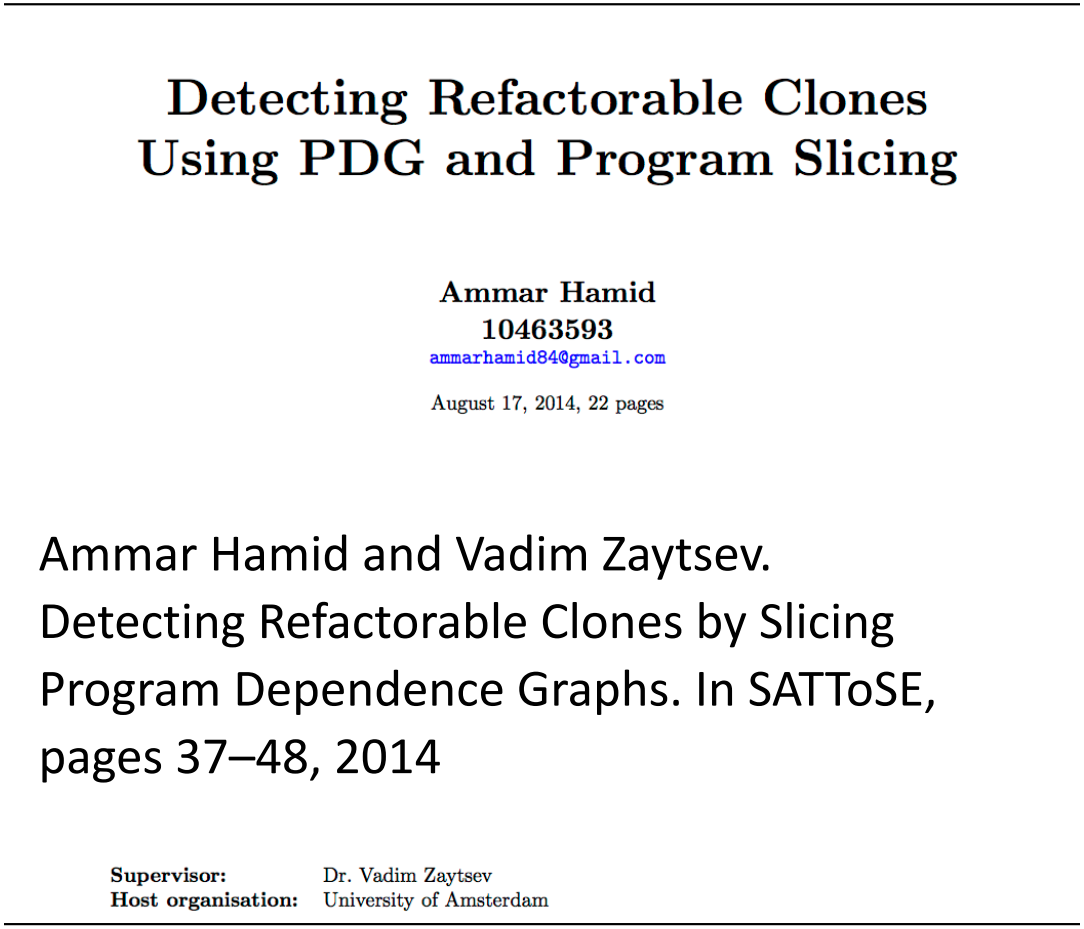
- **Parse.** First parse the code into an abstract-syntax tree

- **Compare ASTs**. Find identical subtrees
  - Optionally ignore the leaves of the tree (e.g., identifiers) to detect Type 2 clones

PDG-based clone detection

- **Extract PDG.** Create a graph that describes control- and data flow dependency (this is hard)
- **Find isomorphic sub-graphs** (this is NP-hard, so you need an approximation)
- You can find (some) Type 3 clones in this way

Graduating on clones



Detecting Refactorable Clones Using PDG and Program Slicing

Ammar Hamid
10463593
ammarhamid84@gmail.com

August 17, 2014, 22 pages

Ammar Hamid and Vadim Zaytsev. Detecting Refactorable Clones by Slicing Program Dependence Graphs. In SATToSE, pages 37–48, 2014

Supervisor: Dr. Vadim Zaytsev
Host organisation: University of Amsterdam

More Code Clones

CodeArena: Inspecting and Improving Code Quality Metrics in Java using Minecraft

Simon Baars
University of Amsterdam
Amsterdam, Netherlands
simon.mailadres@gmail.com

Sander Meester
University of Amsterdam
Amsterdam, Netherlands
sander.meester@student.uva.nl

I. Introduction

Technical debt causes the cost of maintaining a piece a code to rise with it, resulting in more time spend on implementing changes later. Studies show that over 80% of the software development process is spent on software maintenance [6].

There are several factors that influence the maintainability of a codebase, such as code duplication and high code complexity. Various tools and techniques are available that help getting insight into and/or tackling these factors [9, 5, 7]. The functionality of such tools varies from merely informing about the current state of the code, to tools that assist developers in refactoring the code. However, most of these tools do not assist the developer in understanding the causes of factors that influence maintainability. The field of software maintenance and code quality is therefore difficult to make intuitive for developers.

We propose a solution that lets the developer interact with code that does not conform to quality guidelines [4] to gain awareness of what good quality is and how to minimise technical debt.

II. Proposed solution

To make developers aware of harmful coding practises and how they can improve their code, we created CodeArena [1] [2]. CodeArena is an extension to the popular 3D sandbox game called Minecraft. It allows developers to experience the quality of their code and gain progressive insight in the causes of hard-to-maintain code. This tool translates features of a codebase that are considered harmful to monsters in Minecraft, which can then be "fought" to improve the codebase. Fighting the monsters will trace the user back to the source code. If the developer succeeds in solving the issue, the monster will die and the developer will be rewarded in-game. This way, the developer can gradually improve the quality of the code, while learning about code quality in an engaging way.

immediately be refactored, resulting in the code quality being improved.

The developer can traverse different components of the codebase and gain experience points for each metric that has been improved. Points will be subtracted if the metrics are worsened after the developer changes the code. The metrics to be improved consist of code duplication, unit complexity, unit size and unit interface size [4]. Each of these metrics correspond to a different type of monster.

The Java project to be improved can be chosen from disk, or a GitHub URL can be specified to download a project from. After the project has been chosen, the tool will create a monster for each code issue found. The user will have to fix code issues in order to collect experience points.

Fig. 1. Player's view of clones represented as spiders

A. Technical architecture

Our tool is made using Minecraft Forge [2], which provides

Towards Automated Refactoring of Code Clones in Object-Oriented Programming Languages - Work in Progress -

Simon Baars
University of Amsterdam
Amsterdam, Netherlands
simon.mailadres@gmail.com

Ana Oprescu
University of Amsterdam
Amsterdam, Netherlands
AM.Oprescu@uva.nl

Abstract

Duplication in source code can have a major negative impact on the maintainability of source code, as it creates implicit dependencies between fragments of code. Such implicit dependencies often cause bugs and increase maintenance efforts. In this study, we look into the opportunities to automatically refactor these duplication problems for object-oriented programming languages. We propose a method to detect clones that are suitable for refactoring. This method focuses on the context and scope of clones, ensuring our refactoring improves the design and does not create side effects.

Our intermediate results indicate that more than half of the duplication in code is related to each other through inheritance, making it easier to refactor these clones in a clean way. About 40 percent of the duplication can be refactored through method extraction, while other clones require other refactoring techniques or further transformations. Future measurements will provide further insight into what clones should be refactored to improve the design of software systems.

1 Introduction

Duplication in source code is often seen as one the most harmful types of technical debt. In Ma Fowler's "Refactoring" book [Fow99], he claims t *"If you see the same code structure in more than place, you can be sure that your program will be ter if you find a way to unify them."*. Bruntinl al. [BVDVET05] show that code clones can contrib up to 25% of the code size, which has a negative pact on the maintainability.

Refactoring is used to improve the quality-rela attributes of a codebase (maintainability, performa etc.) without changing the functionality. Many m ods were introduced to aid the process of refac ing [Fow99, Wak04], and are integrated into most m ern IDE's. However, most of these methods still quire a manual assessment of where and when to ply them. This means refactoring is either a sig cant part of the development process [LST78, MT or does not happen at all [MVD+03]. For a large p proper refactoring requires domain knowledge. H ever, there are also refactoring opportunities that rather trivial and repetitive to execute. Our goa investigating to what extend code clones can be au matically refactored.

A survey by Roy et al. [RC07] describes var ways in which clones can be identified. Most cl detection tools focus on finding clones that align w these definitions. In this paper, we outline challe with these clone type definitions when considere a refactoring context. We next propose solution these problems that would enable the detectio clones that can and should be refactored, rather t fragments of code that are just similar.

Clone visualization – file view

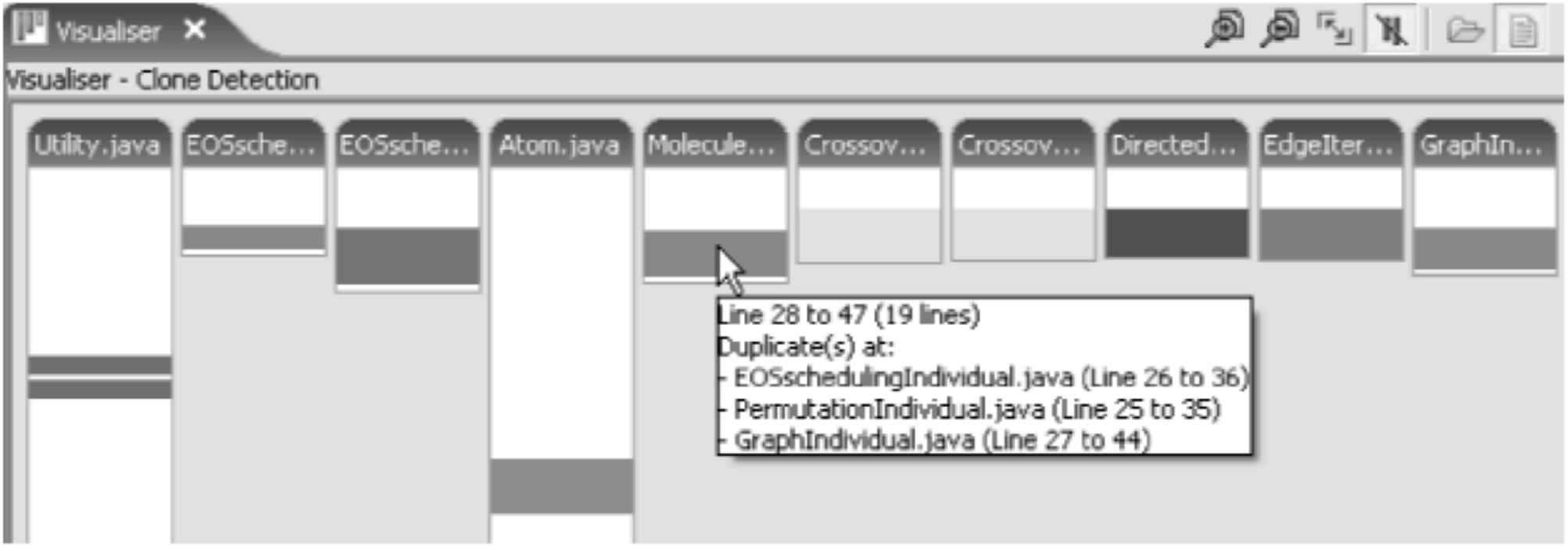
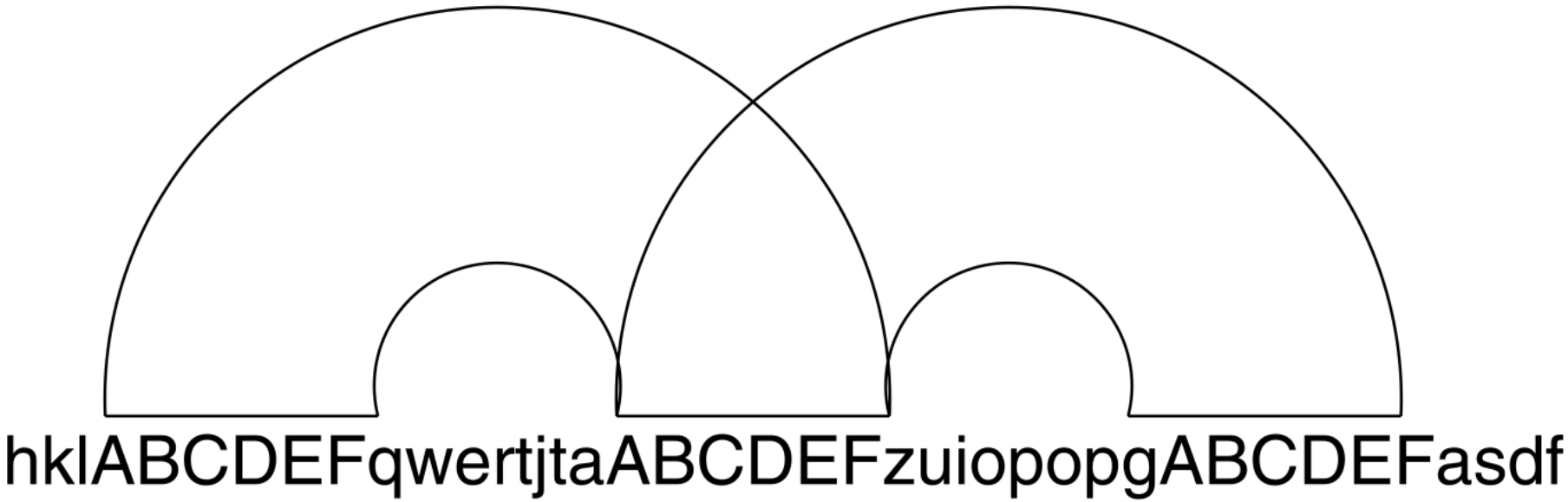


Fig. 2.8. Clones visualizer view in Eclipse adapted from [489]

Source: R. Koschke, *Identifying and Removing Software Clones,* in *Software Evolution*, Springer, 2008.

Clone visualization – arc diagrams



Source: R. Koschke, *Identifying and Removing Software Clones,* in *S*oftware Evolution, Springer, 2008.

Clone visualization – friend wheel

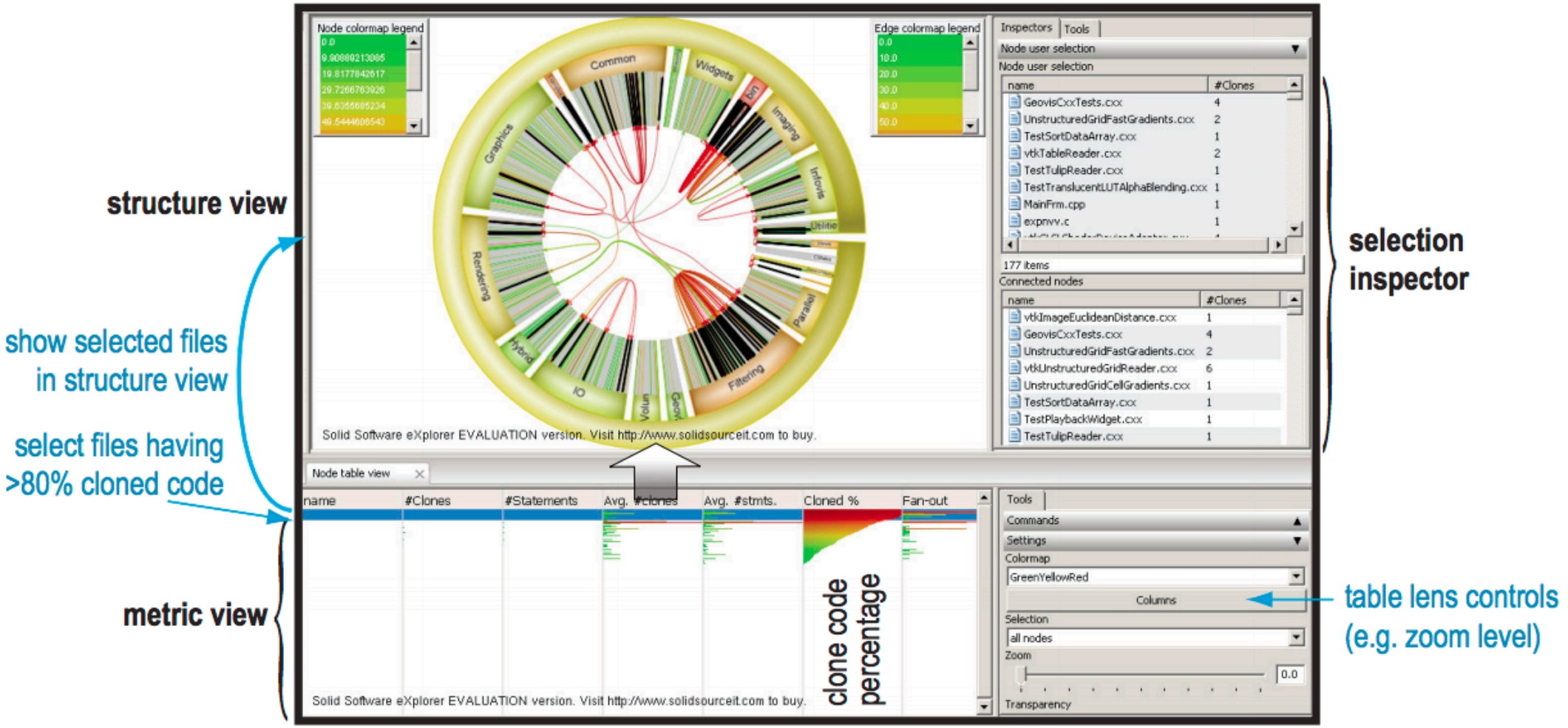


Figure 2. SolidSDD metric view (bottom) and structure view (top) with high-clone percentage files highlighted (see Sec. II).

Source: L. Voinea and A. Telea, *Visual Clone Analysis with SolidSDD,* in Proceedings of IEEE VISSOFT, 2014.

Clone visualization – tree maps

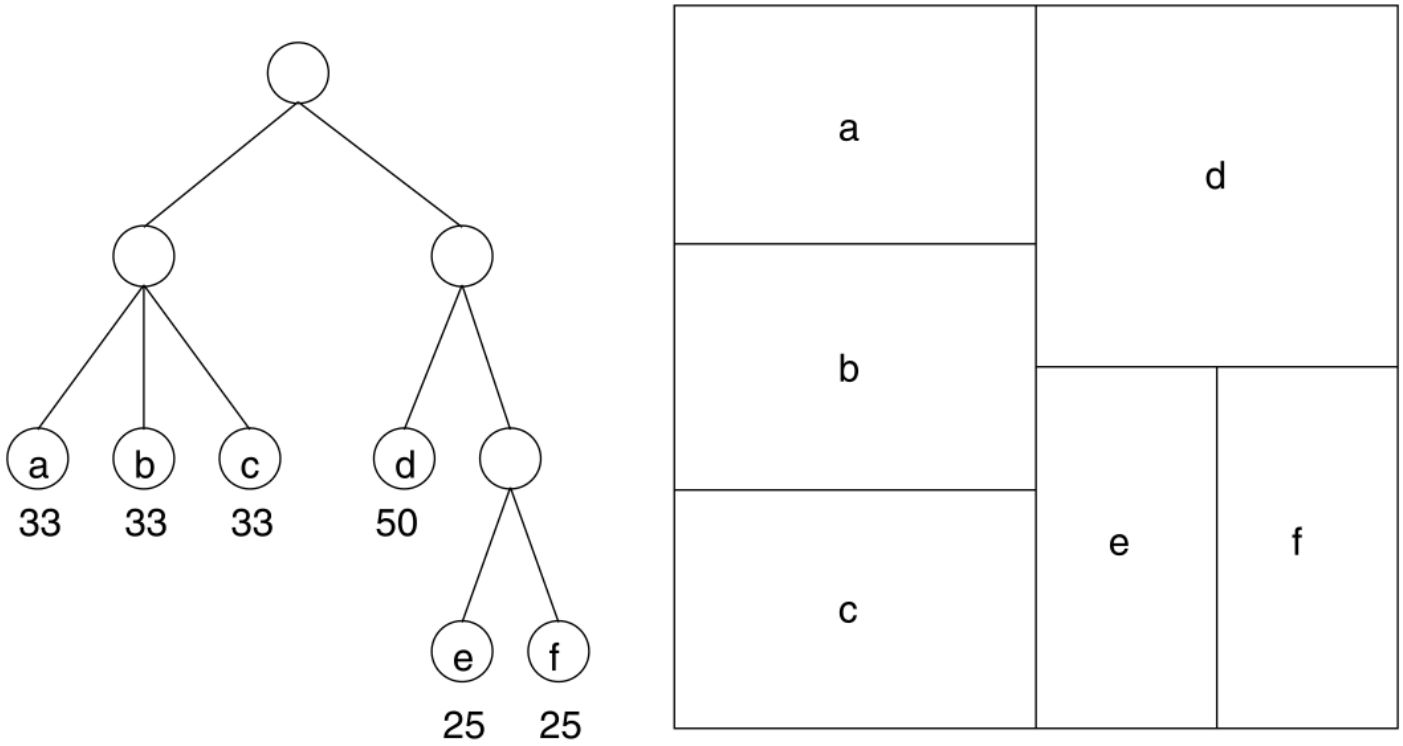


Fig. 2.6. A system decomposition whose leaves are annotated with the number of cloned lines of code and its corresponding tree map

Source: R. Koschke, *Identifying and Removing Software Clones,* in Software Evolution, Springer, 2008.

Clone visualization – polymetric views

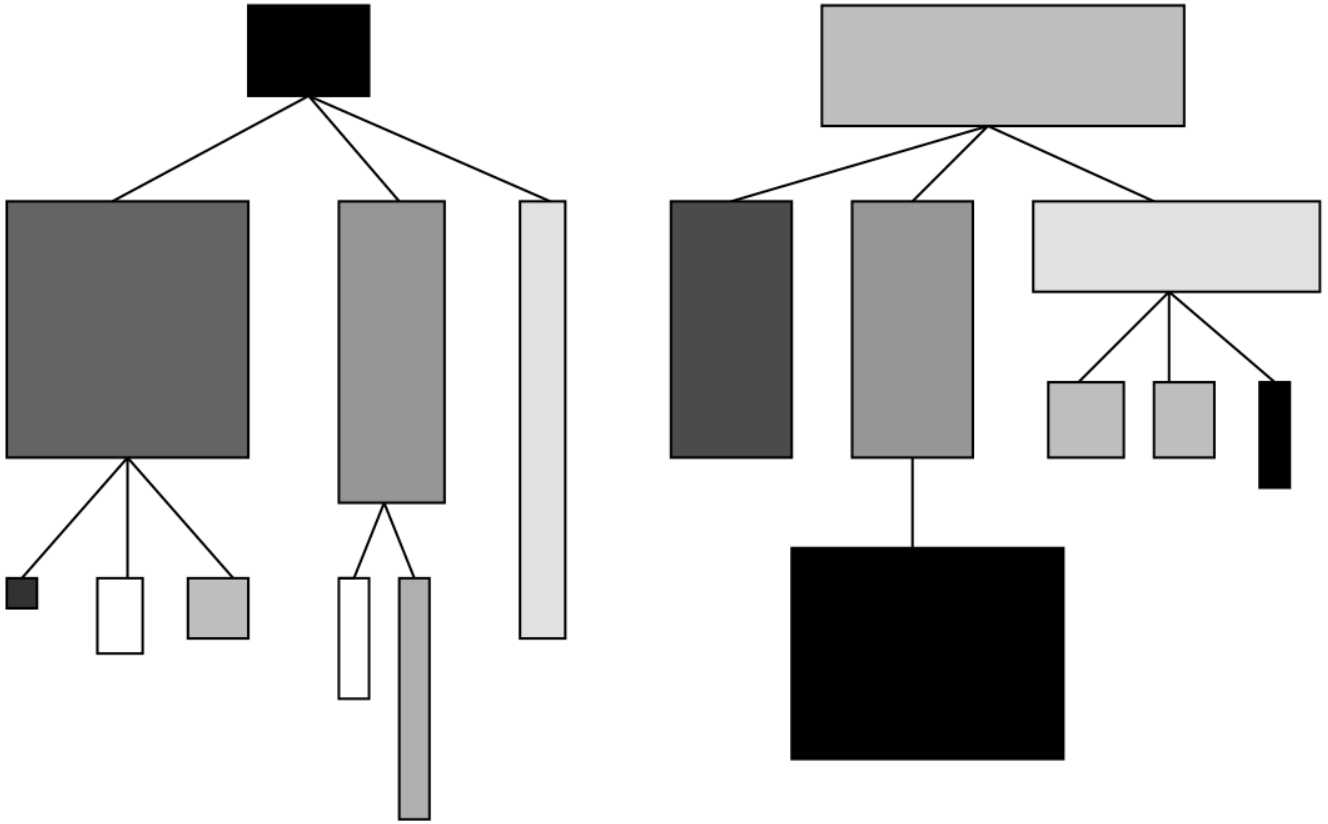
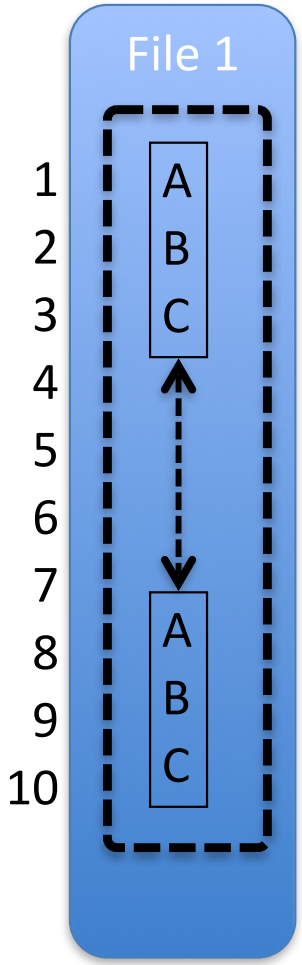


Fig. 2.5. Polymetric view adapted from [437]

Source: R. Koschke, *Identifying and Removing Software Clones,* in *Software Evolution*, Springer, 2008.

Clone class subsumption



Clones:

1. File 1: 1-3
2. File 1: 8-10

Assuming Type 1 or 2

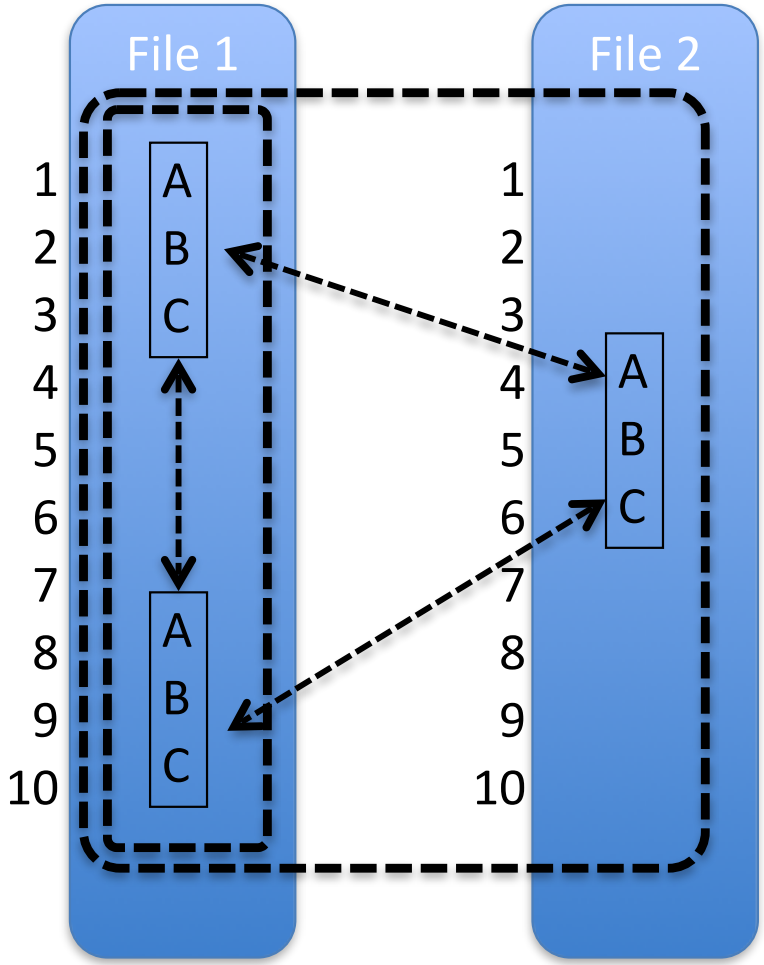
Clone pairs:

1. File 1: 1-3 <-> File 1: 8-10

Clone classes:

1. File 1: 1-3, File 1: 8-10

Clone class subsumption



Assuming Type 1 or 2

Clones:

1. File 1: 1-3
2. File 1: 8-10
3. File 2: 4-6

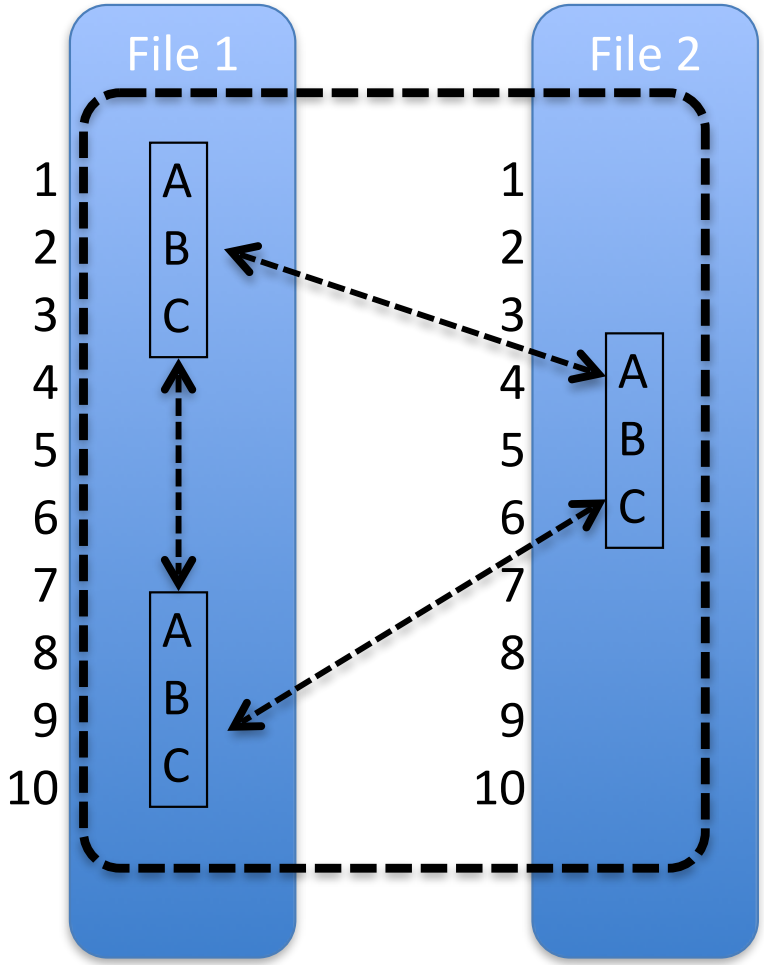
Clone pairs:

1. File 1: 1-3 <-> File 1: 8-10
2. File 1: 1-3 <-> File 2: 4-6
3. File 1: 8-10 <-> File 2: 4-6

Clone classes:

1. File 1: 1-3, File 1: 8-10
2. File 1: 1-3, File 1: 8-10, File 2: 4-6

Clone class subsumption



Assuming Type 1 or 2

Clones:

1. File 1: 1-3
2. File 1: 8-10
3. File 2: 4-6

Clone pairs:

1. File 1: 1-3 <-> File 1: 8-10
2. File 1: 1-3 <-> File 2: 4-6
3. File 1: 8-10 <-> File 2: 4-6

Clone classes:

1. ~~File 1: 1-3, File 1: 8-10~~
2. File 1: 1-3, File 1: 8-10, File 2: 4-6

The plan for today

1. Clone detection
   - Finding clones
   - Visualizing clones
2. Break (15 minutes max.)
3. Clone management
   - Preventive
   - Compensative
   - Corrective
4. Lab series 2

The plan for today

1. Clone detection
   - Finding clones
   - Visualizing clones

2. Break (15 minutes max.)

3. Clone management
   - Preventive
   - Compensative
   - Corrective

4. Lab series 2

Code clones – good, bad, or ugly?

Picture copyright chillyfraco @deviantART

Causes of cloning

?

Causes of cloning

- Forced by programming language
- Forced by software design
- Delayed refactoring (due to uncertainty or time pressure)
- Idiomatic solutions or coding standards
- Templating
- Forking

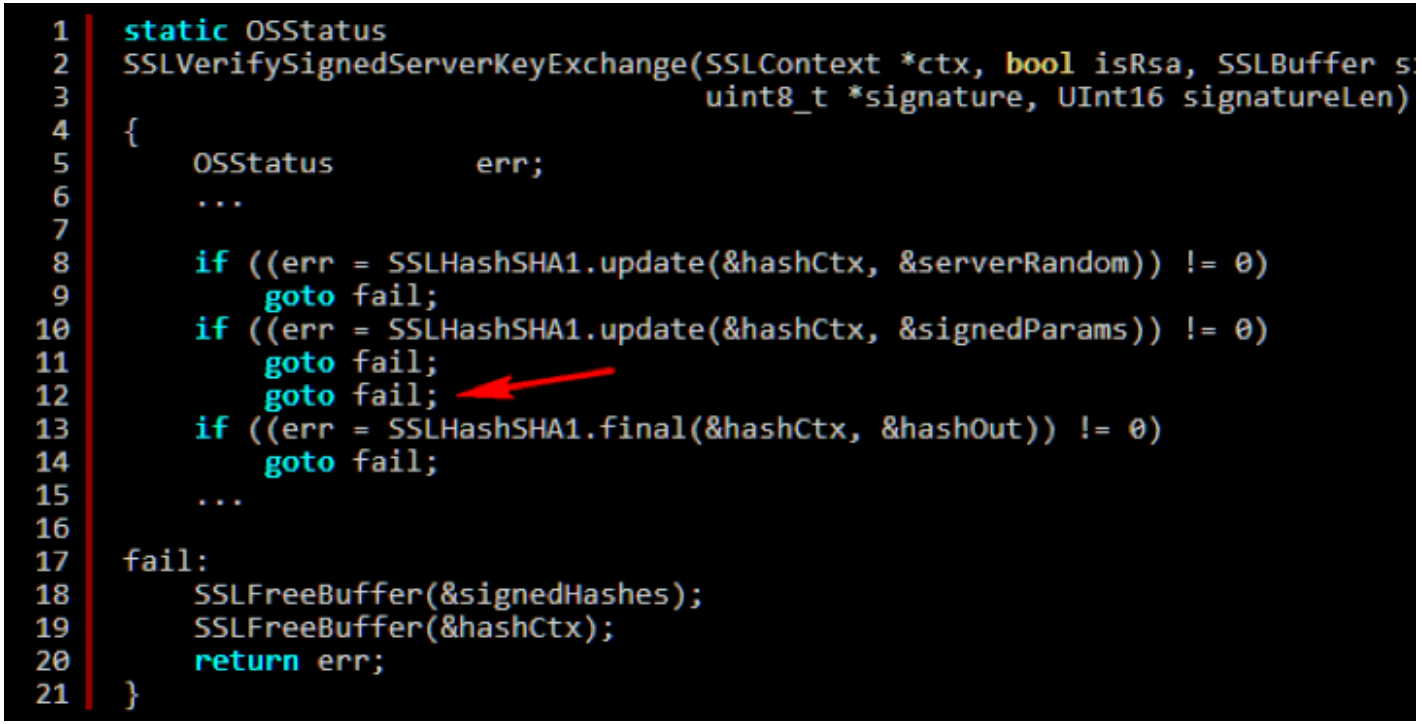
```
namespace MM.Model
{
  public class Program : Definition
  {
    override public void accept (Visitor
visitor)
    {
      visitor.visit(this);
    }
  }
}
```

Consequences of cloning

?

Consequences of cloning

- Increased maintenance effort?
- Increased productivity during development?
- Decreased risk of breaking things?



```
1  static OSStatus
2  SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer s
3                                   uint8_t *signature, UInt16 signatureLen)
4  {
5      OSStatus        err;
6      ...
7
8      if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
9          goto fail;
10     if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
11         goto fail;
12         goto fail;
13     if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
14         goto fail;
15     ...
16
17 fail:
18     SSLFreeBuffer(&signedHashes);
19     SSLFreeBuffer(&hashCtx);
20     return err;
21 }
```

Clone management

Preventive – no clones in the first place

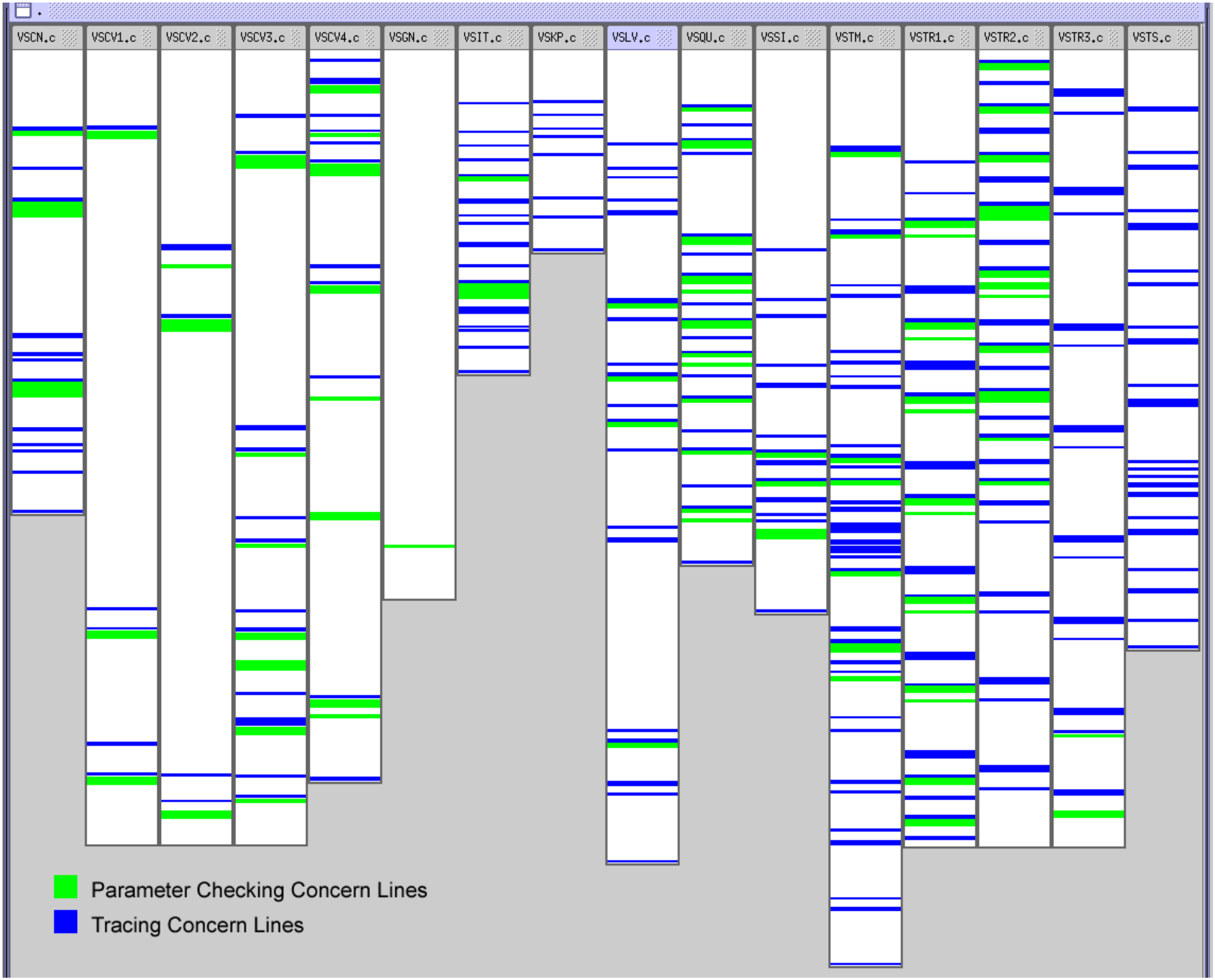
Compensative – coping with existing clones

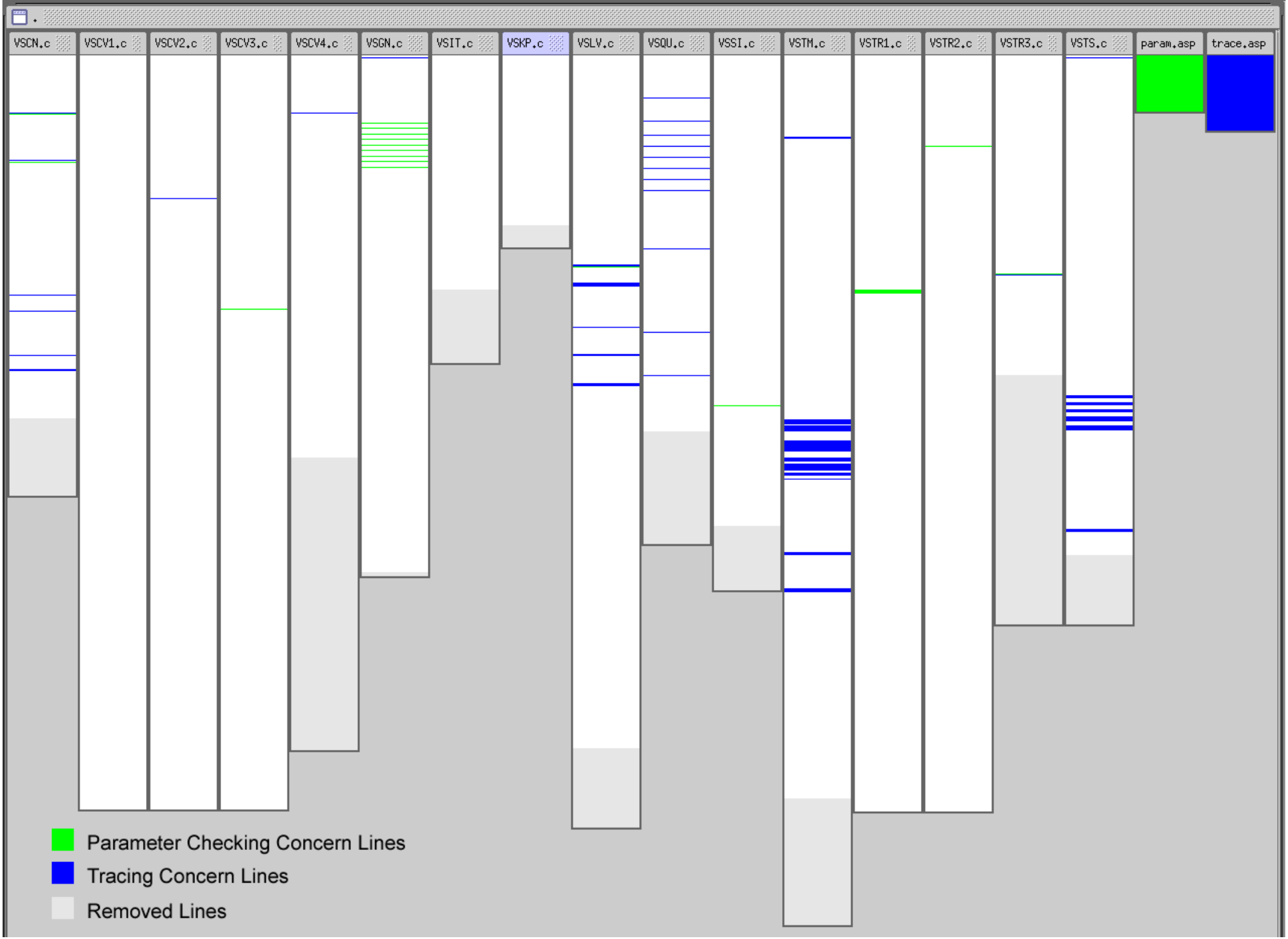
Corrective – fixing existing clones

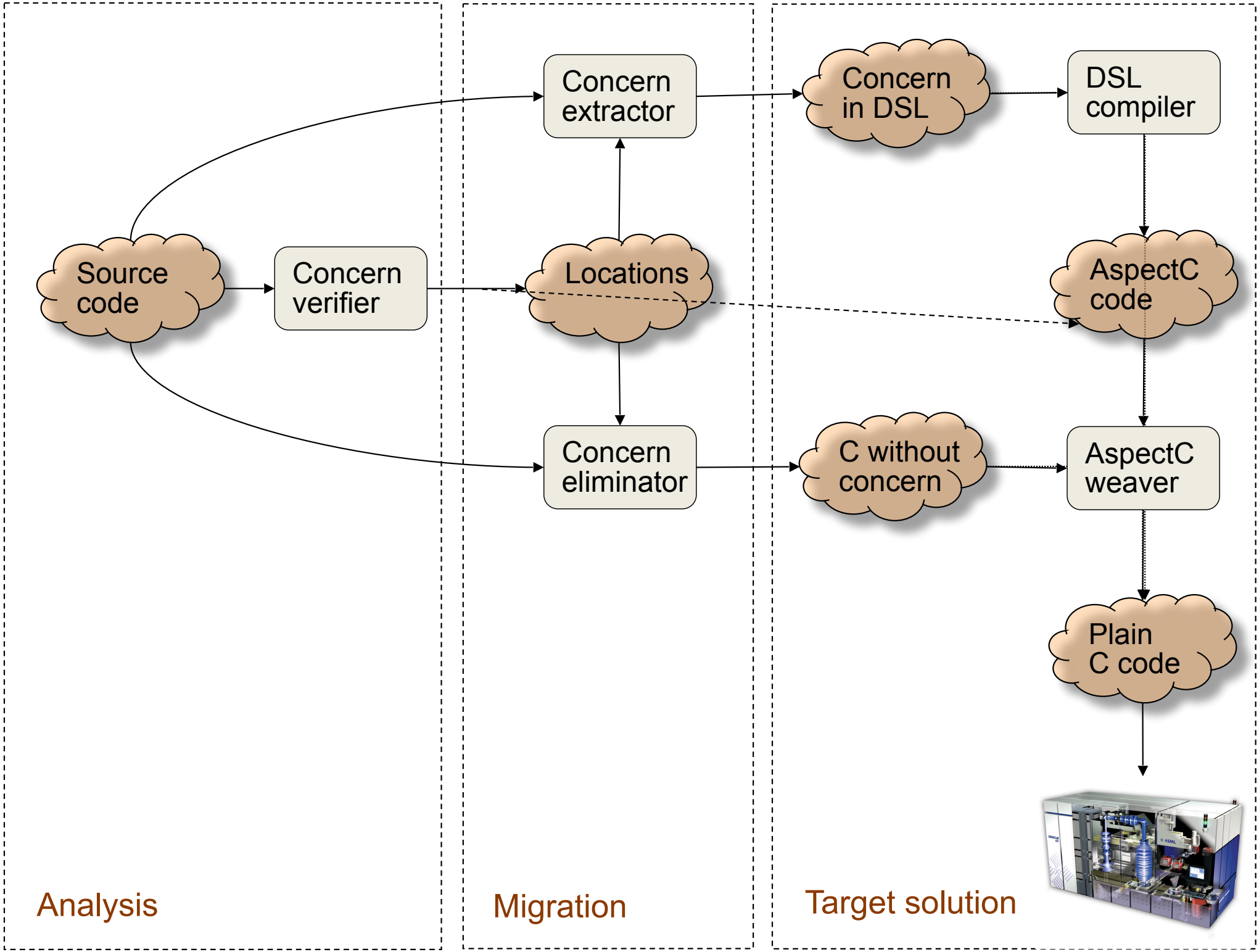
Corrective clone management

Refactor clones into something “better”

- Example: refactor clone into a method and replace clones by two method calls
- More elaborate: apply aspect-oriented programming


VSCN.c
VSCV1.c
VSCV2.c
VSCV3.c
VSCV4.c
VSGN.c
VSIT.c
VSKP.c
VSLV.c
VSQU.c
VSSI.c
VSTM.c
VSTR1.c
VSTR2.c
VSTR3.c
VSTS.c
Parameter Checking Concern Lines
Tracing Concern Lines


VSCN.c
VSCV1.c
VSCV2.c
VSCV3.c
VSCV4.c
VSGN.c
VSIT.c
VSKP.c
VSLV.c
VSQU.c
VSSI.c
VSTM.c
VSTR1.c
VSTR2.c
VSTR3.c
VSTS.c
param.asp
trace.asp
Parameter Checking Concern Lines
Tracing Concern Lines
Removed Lines


Concern extractor
Concern in DSL
DSL compiler
Source code
Concern verifier
Locations
AspectC code
Concern eliminator
C without concern
AspectC weaver
Plain C code
Analysis
Migration
Target solution

Preventive clone management

“Disable CTRL-C CTRL-V” ;-)

While typing, or at check-in, check for clones

Prerequisites:

- Organizational support needed
- Programming technology must allow non-cloned solutions

Compensative clone management

Maintain knowledge of clones (in the IDE)

When developers change code, they are alerted that clones exist

Clones can be annotated with additional knowledge, etc.

The plan for today

1. Clone detection
   - Finding clones
   - Visualizing clones

2. Break (15 minutes max.)

3. Clone management
   - Preventative
   - Compensative
   - Corrective

4. Lab series 2