An Empirical Study of Code Clone Genealogies

Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy
University of Washington
University of British Columbia

ESEC/FSE Sept 2005

Conventional Wisdom

*Code clones indicate **bad smells** of poor design. We must **aggressively refactor** clones.*

```
public void updateFrom  (Class c )  {
      String cType = Util.makeType(c.Name());
      if (seenClasses.contains(cType)) {
         return;
      }
      seenClasses.add(cType);
      if (hierarchy != null) {
         ....
      }
      ...
```

```
public void updateFrom  (ClassReader cr )  {
      String cType =CTD.convertType (c.Name());
      if (seenClasses.contains(cType)) {
         return;
      }
      seenClasses.add(cType);
      if (hierarchy != null) {
       ....
      }
      ...
```

Our Previous Study of Copy and Paste Programming Practices at IBM

[Kim et al. ISESE2004]

- Even skilled programmers often **create and manage** code clones with clear intent.
  - Programmers cannot refactor clones because of **programming language limitations.**
  - Programmers **keep** and **maintain clones** until they realize how to abstract the common part of clones.
  - Programmers often **apply similar changes** to clones.

Research Questions

How do clones evolve over time?

- consistently changed?
- long-lived (or short-lived)?
- easily refactorable?

Previous Studies of Code Clones

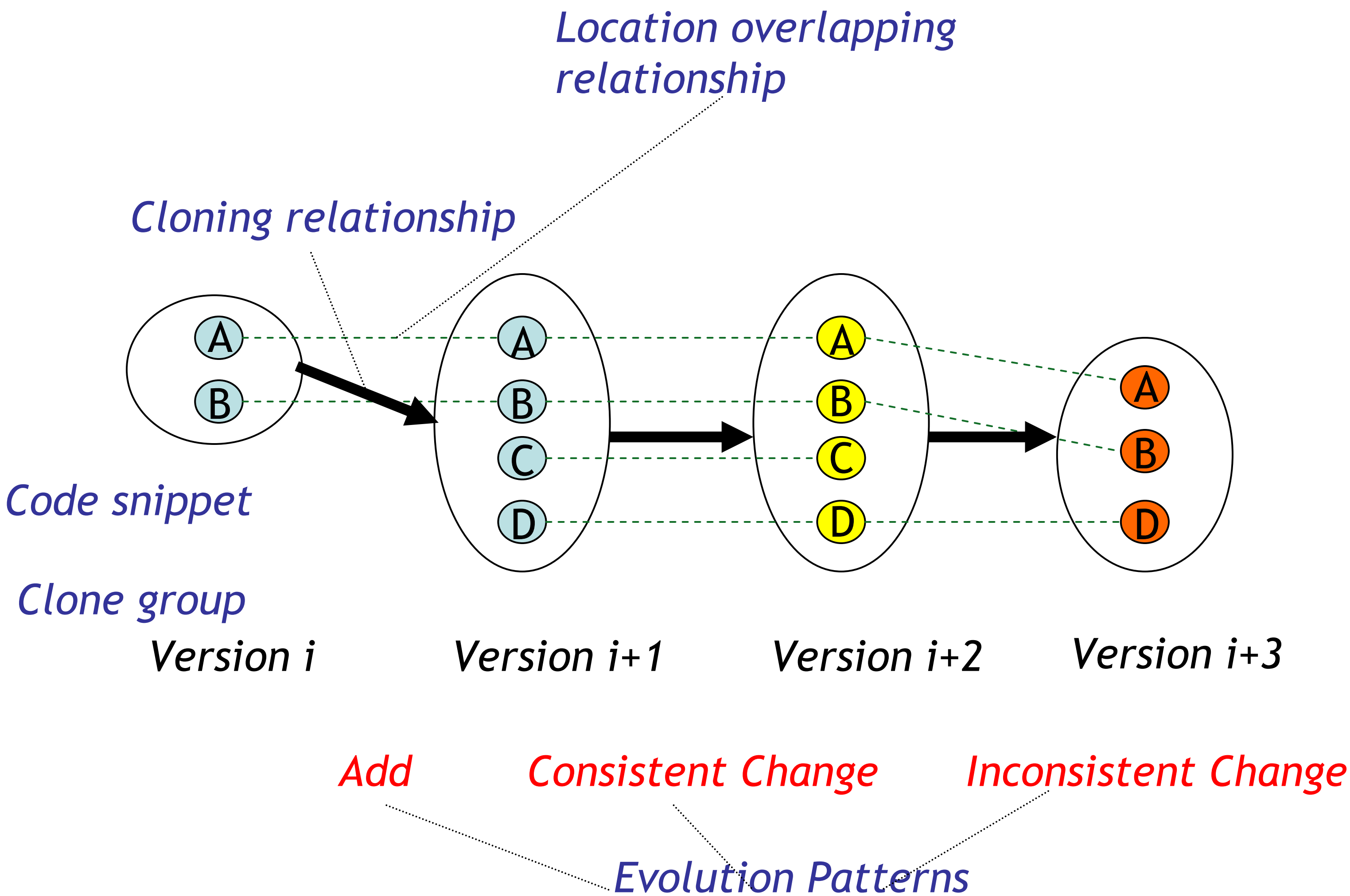
- automatic clone detection
  - lexical, syntactic (AST or PDG), metric, etc.
- studies of clone coverage ratio
  - gcc (8.7%), JDK (29%), Linux (22.7%), etc.
- studies of clone coverage change
  - changes of clone coverage in Linux [Antoniol+02], [Li+04]

These studies do not answer how individual clones changed with respect to other clones.

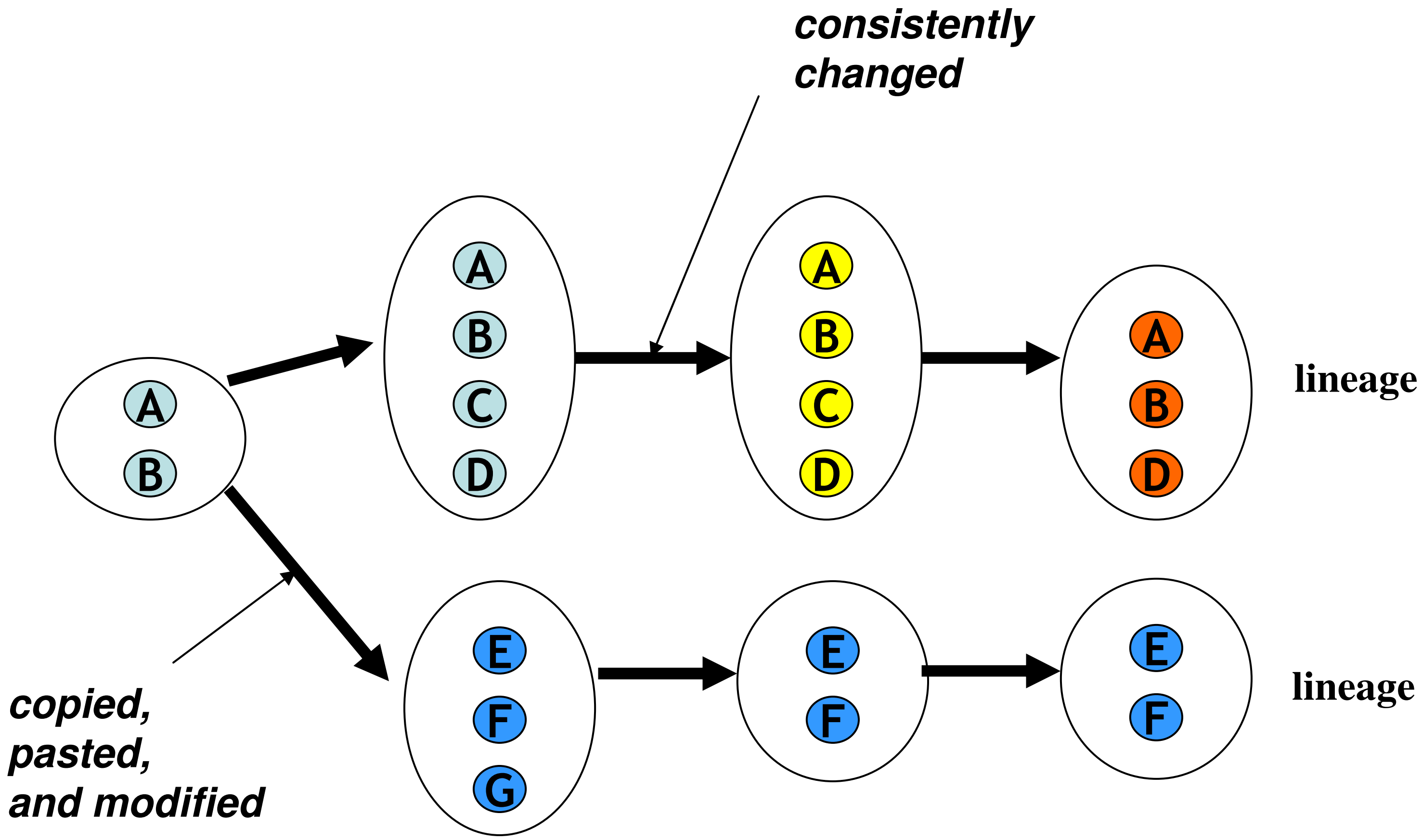
Outline

- motivation
- clone genealogy : model and tool
- study procedure and results

Model of Clone Evolution



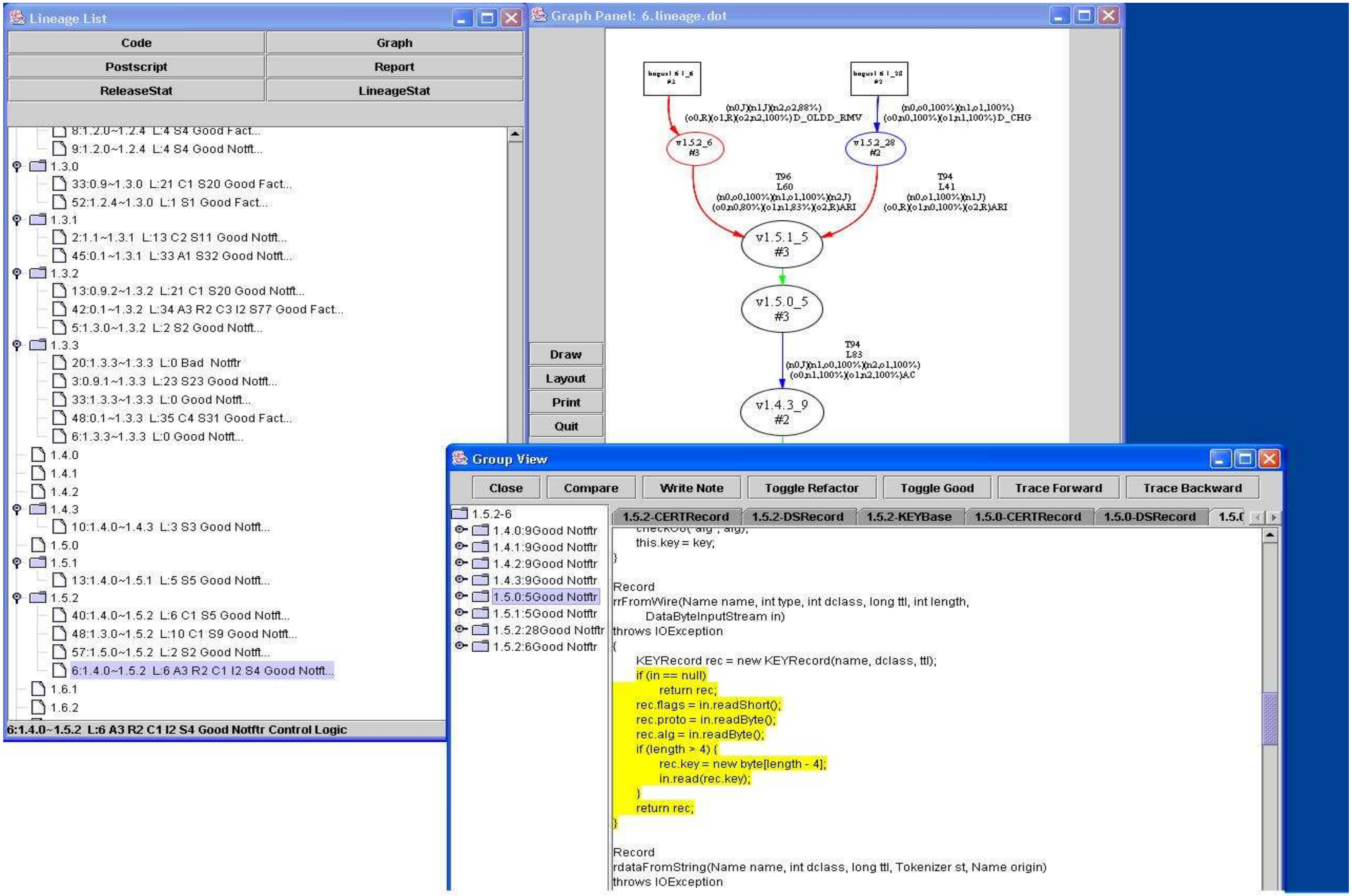
Clone genealogy is a set of clone groups connected by cloning relationships over time.



Clone Genealogy Extractor (CGE)

Given multiple versions of a program, V_k for $1 \leq k \leq n$.

- find clone groups in each version using CCFinder.
- find cloning relationships among clone groups of V_i and V_{i+1} using CCFinder.
- map clones of V_i and V_{i+1} using diff based algorithm.
- separate each connected component of cloning relationships (a clone genealogy).
- identify clone evolution patterns in each genealogy.


Lineage List
Code
Graph
Postscript
Report
ReleaseStat
LineageStat
8:1.2.0~1.2.4 L:4 S4 Good Fact...
9:1.2.0~1.2.4 L:4 S4 Good Notft...
1.3.0
33:0.9~1.3.0 L:21 C1 S20 Good Fact...
52:1.2.4~1.3.0 L:1 S1 Good Fact...
1.3.1
2:1.1~1.3.1 L:13 C2 S11 Good Notft...
45:0.1~1.3.1 L:33 A1 S32 Good Notft...
1.3.2
13:0.9.2~1.3.2 L:21 C1 S20 Good Notft...
42:0.1~1.3.2 L:34 A3 R2 C3 I2 S77 Good Fact...
5:1.3.0~1.3.2 L:2 S2 Good Notft...
1.3.3
20:1.3.3~1.3.3 L:0 Bad Notftr
3:0.9.1~1.3.3 L:23 S23 Good Notft...
33:1.3.3~1.3.3 L:0 Good Notft...
48:0.1~1.3.3 L:35 C4 S31 Good Fact...
6:1.3.3~1.3.3 L:0 Good Notft...
1.4.0
1.4.1
1.4.2
1.4.3
10:1.4.0~1.4.3 L:3 S3 Good Notft...
1.5.0
1.5.1
13:1.4.0~1.5.1 L:5 S5 Good Notft...
1.5.2
40:1.4.0~1.5.2 L:6 C1 S5 Good Notft...
48:1.3.0~1.5.2 L:10 C1 S9 Good Notft...
57:1.5.0~1.5.2 L:2 S2 Good Notft...
6:1.4.0~1.5.2 L:6 A3 R2 C1 I2 S4 Good Notft...
1.6.1
1.6.2
6:1.4.0~1.5.2 L:6 A3 R2 C1 I2 S4 Good Notftr Control Logic
Graph Panel: 6.lineage.dot
Draw
Layout
Print
Quit
v1.5.2_6
#3
v1.5.2_28
#2
T96
L60
T94
L41
v1.5.1_5
#3
v1.5.0_5
#3
T94
L83
v1.4.3_9
#2
Group View
Close
Compare
Write Note
Toggle Refactor
Toggle Good
Trace Forward
Trace Backward
1.5.2-6
1.4.0:9Good Notftr
1.4.1:9Good Notftr
1.4.2:9Good Notftr
1.4.3:9Good Notftr
1.5.0:5Good Notftr
1.5.1:5Good Notftr
1.5.2:28Good Notftr
1.5.2:6Good Notftr
1.5.2-CERTRecord
1.5.2-DSRecord
1.5.2-KEYBase
1.5.0-CERTRecord
1.5.0-DSRecord
this.key = key;
}
Record
rrFromWire(Name name, int type, int dclass, long ttl, int length,
DataByteInputStream in)
throws IOException
{
KEYRecord rec = new KEYRecord(name, dclass, ttl);
if (in == null)
return rec;
rec.flags = in.readShort();
rec.proto = in.readByte();
rec.alg = in.readByte();
if (length > 4) {
rec.key = new byte[length - 4];
in.read(rec.key);
}
return rec;
}
Record
rdataFromString(Name name, int dclass, long ttl, Tokenizer st, Name origin)
throws IOException

Outline

- motivation
- clone genealogy : model and tool
- q study procedure and results

Two Java Subject Programs

Program	*carol*	*dnsjava*
LOC	7878 ~ 23731	5756 ~ 21188
Duration	2 years 2 months	5 years 8 months
versions	37	224

versions: a set of check-in snapshots that increased or decreased the total lines of code clones

Running CGE on Java Programs

- CCFinder setting
  - minimum token length = 30
  - longest sequence matching
- CGE setting
  - text similarity threshold = 0.3
- false positives
  - repetitive field declaration
  - repetitive static method invocation
  - a series of case switch statements
  - etc.

Consistently Changing Clones

Question: How often do programmers update clones consistently?

Study Method:

- A genealogy has a "*consistent change*" pattern iff all lineages include at least one consistent change pattern.
- We counted genealogies with a "*consistent change*" pattern.

Consistently Changing Clones

Results:

- 38% and 36% of genealogies include a *consistent change* pattern.

> **Consistent with conventional wisdom, programmers often apply similar changes repetitively to clones.**

Volatile Clones

Question: How long do clones survive in the system before they disappear, and how do they disappear?

Study Method:

- A genealogy is *"dead"* if it does not include clones of the final version.
- We measured the age (lifespan or length) of dead genealogies.

Volatile Clones

Results:

disappeared within	*carol*	*dnsjava*
2 versions	52%	35%
5 versions	75%	36%
10 versions	79%	48%

- 26% and 34% of clone lineages were discontinued because of divergent changes in the clone group.

How do lineages disappear?

reasons	*carol*	*dnsjava*
divergent changes	26%	34%
refactoring or removal	67%	45%
cut off by the threshold	7%	21%

Contrary to conventional wisdom, immediate refactoring may be unnecessary or counterproductive <u>in some cases</u>.

Locally Unfactorable Clones

Question: How many clones are difficult to refactor?

Study Method:

- A clone group is locally unfactorable if
  - programmers cannot use standard refactoring techniques, or
  - programmer must deal with cascading non-local changes, or
  - programmers cannot remove duplication due to programming language limitations.
- We manually inspected all genealogies and counted locally unfactorable genealogies.

Locally Unfactorable Clones

```
public void exportObject(Remote obj)
throws RemoteException{
  if (TraceCarol.isDebugRmiCarol()) {
    TraceCarol.debugRmiCarol(
      "MultiPRODelegate.exportObject(" ... .
  }
  try {
    if (init) {
     for (Enumeration e =
      activePtcls.elements(); ...

      ((ObjDlgt)e.nextElement()).exportObject
      (obj);
      }
    }
  }catch (Exception e) {
    String msg = "exportObject(Remote obj)
      fail";
    TraceCarol.error(msg,e);
    throw new RemoteException(msg);
  }
```

```
public void unexportObject(Remote obj)
throws NoSuchObjectException {
  if (TraceCarol.isDebugRmiCarol()) {
    TraceCarol.debugRmiCarol(
      "MultiPRODelegate.unexportObject(" ... .
  }
  try {
    if (init) {
     for (Enumeration e =
      activePtcls.elements(); ...

      ((ObjDlgt)e.nextElement()).unexportObje
      ct(obj);
      }
    }
  } catch (Exception e) {
    String msg = "unexportObject(Remote obj)
      fail";
    TraceCarol.error(msg,e);
    throw new NoSuchObjectException(msg);
  }
```

Locally Unfactorable Clones

Result:

- 64% and 49% of genealogies are locally unfactorable.

Contrary to conventional wisdom, refactoring may not remove many clones easily.

Long-Lived Clones

Question: For clones that live for a long time and tend to change with other clones, can they be easily refactored?

Study Method:

- We measured cumulative proportion of locally unfactorable and consistently changed genealogies.

Long-Lived Clones

Results:

- 51% and 61% of genealogies that lasted more than half of programs' lifetime are locally unfactorable and consistently changing.
- The proportion of locally unfactorable yet consistently changed genealogies increases with the age of genealogies.

Contrary to conventional wisdom, refactoring cannot help many consistently changed, long-lived clones.

Study Limitations

- clone detection techniques
  - CCFinder vs. other clone detection techniques.
- location tracking techniques
  - diff vs. other location tracking techniques.
- subject programs
  - 20KLOC vs. large scale projects
- time granularity
  - versions vs. editing operations
- language dependency
  - Java vs. other languages

Summary

- We have built a tool that extracts history of code clones from a set of program versions.
- Our study of clone genealogy contradicts some conventional wisdom about code clones.
  - Immediate and aggressive refactoring may be unnecessary for volatile and diverging clones.
  - Refactoring may not help many long-lived and consistently changing clones.
- Our study opens up opportunities for complementary clone maintenance tools.