All we like sheep:
Cloning as a software engineering tool

Michael W. Godfrey
University of Waterloo

Overview

- What are code clones?
  - Some motivating examples
  - Kinds of clones, by structure
- How do we detect clones?
- Just how bad are clones? How do we know?
  - A taxonomy of clones, by design intent

Some examples of code clones

Consider this code…

```
const char *err = ap_check_cmd_context(cmd, GLOBAL_ONLY);
if (err != NULL) {
   return err;
}
ap_threads_per_child = atoi(arg);
if (ap_threads_per_child > thread_limit) {
   ap_log_error(APLOG_MARK, APLOG_STARTUP, 0, NULL,
               "WARNING: ThreadsPerChild of %d exceeds ThreadLimit "
               "value of %d", ap_threads_per_child,
               thread_limit);
   ....
   ap_threads_per_child = thread_limit;
}
else if (ap_threads_per_child < 1) {
   ap_log_error(APLOG_MARK, APLOG_STARTUP, 0, NULL,
               "WARNING: Require ThreadsPerChild > 0, setting to 1");
   ap_threads_per_child = 1;
}
return NULL;
```

and this code …

```
const char *err = ap_check_cmd_context(cmd, GLOBAL_ONLY);
if (err != NULL) {
    return err;
}
ap_threads_per_child = atoi(arg);
if (ap_threads_per_child > thread_limit) {
    ap_log_error(APLOG_MARK, APLOG_STARTUP, 0, NULL,
                 "WARNING: ThreadsPerChild of %d exceeds ThreadLimit "
                 "value of %d threads,", ap_threads_per_child,
                 thread_limit);
    ....
    ap_threads_per_child = thread_limit;
}
else if (ap_threads_per_child < 1) {
        ap_log_error(APLOG_MARK, APLOG_STARTUP, 0, NULL,
                 "WARNING: Require ThreadsPerChild > 0, setting to 1");
        ap_threads_per_child = 1;
}
return NULL;
```

… or these two functions

```
static GnmValue *
gnumeric_oct2bin (FunctionEvalInfo *ei, GnmValue const * const *argv)
{
   return val_to_base (ei, argv[0], argv[1],
        8, 2,
        0, GNM_const(7777777777.0),
        V2B_STRINGS_MAXLEN | V2B_STRINGS_BLANK_ZERO);
}

static GnmValue *
gnumeric_hex2bin (FunctionEvalInfo *ei, GnmValue const * const *argv)
{
   return val_to_base (ei, argv[0], argv[1],
        16, 2,
        0, GNM_const(9999999999.0),
        V2B_STRINGS_MAXLEN | V2B_STRINGS_BLANK_ZERO);
}
```

Or this …

```
static PyObject *
py_new_RangeRef_object (const GnmRangeRef *range_ref){
  py_RangeRef_object *self;
  self = PyObject_NEW py_RangeRef_object,
      &py_RangeRef_object_type);
  if (self == NULL) {
      return NULL;
  }
  self->range_ref = *range_ref;
  return (PyObject *) self;
}
```

… and this

```
static PyObject *
py_new_Range_object (GnmRange const *range) {
  py_Range_object *self;
  self = PyObject_NEW (py_Range_object,
      &py_Range_object_type);
  if (self == NULL) {
      return NULL;
  }
  self->range = *range;
  return (PyObject *) self;
}
```

An overview of clone detection

What's a clone?

"Software clones are segments of code that are similar according to some definition of similarity."

– Ira Baxter, 2002

- No universally agreed upon definition
- Often use "what my tool found" as ground truth
  - Algorithms, thresholds may vary greatly
  - Could hand examine subset of results to guess false positive rate
  - False negatives? … and no ground truth from experts typically.
- Hard to compare results!

Bellon's taxonomy

Type 1 Program text (i.e., token stream) identical
… but white space / comments may differ

Type 2 … and literals + identifiers may be different

Type 3 … and gaps allowed (can add/delete sections)

Type 4 Two code segments have same semantics
(Undecidable in general, not sought often)

- There are other kinds of "clones" that don't fit well here
- Note that type 1, 2, and 4 clones form equivalence classes, but type 3 clones do not

Bellon's taxonomy

- Type 1 clones are fairly easy to detect
  - Tokenize the source code, remove comments
  - Simple approach:

```
% tokenize file1.c > f1.c
% tokenize file2.c > f2.c
% diff -w f1.c f2.c
```

  - Scalable approach:
    - Progressively build a suffix tree / array to store *all* known partial sequences of tokens

Bellon's taxonomy

- Type 2 clones are almost as easy
  - Extra step in tokenization:
    - All identifiers mapped to special token <ID>
    - All explicit string values mapped to <STRING>
    - All explicit numerical values mapped to <NUM>

Bellon's taxonomy

- Type 3 clones
  - Look for type 2 clones, but allow "gaps" up to some threshold of lines/tokens
  - Notes:
    - Given a big enough threshold, any two pieces of code are type 3 clones!
    - *"is-a-type-3-clone-of"* is not transitive

Bellon's taxonomy

- Type 4 (semantically similar) clones
  - *"Does P1 have same semantics as P2"* is undecidable in the general case
  - Typically not done, no general purpose detector exists
    - Type 4 category is included for sake of completeness
  - But if we are interested, we can make guesses using various tricks

    e.g., common test suites, dynamic traces, NLP analysis of comments

Spot the clone type!

```
const char *err = ap_check_cmd_context(cmd, GLOBAL_ONLY);
if (err != NULL) {
   return err;
}
ap_threads_per_child = atoi(arg);
if (ap_threads_per_child > thread_limit) {
   ap_log_error(APLOG_MARK, APLOG_STARTUP, 0, NULL,
               "WARNING: ThreadsPerChild of %d exceeds ThreadLimit "
               "value of %d", ap_threads_per_child,
               thread_limit);
   ....
   ap_threads_per_child = thread_limit;
}
else if (ap_threads_per_child < 1) {
   ap_log_error(APLOG_MARK, APLOG_STARTUP, 0, NULL,
               "WARNING: Require ThreadsPerChild > 0, setting to 1");
   ap_threads_per_child = 1;
}
return NULL;
```

Spot the clone type!

```
const char *err = ap_check_cmd_context(cmd, GLOBAL_ONLY);
if (err != NULL) {
    return err;
}
ap_threads_per_child = atoi(arg);
if (ap_threads_per_child > thread_limit) {
    ap_log_error(APLOG_MARK, APLOG_STARTUP, 0, NULL,
                 "WARNING: ThreadsPerChild of %d exceeds ThreadLimit "
                 "value of %d threads,", ap_threads_per_child,
                 thread_limit);
    ....
    ap_threads_per_child = thread_limit;
}
else if (ap_threads_per_child < 1) {
        ap_log_error(APLOG_MARK, APLOG_STARTUP, 0, NULL,
                 "WARNING: Require ThreadsPerChild > 0, setting to 1");
        ap_threads_per_child = 1;
}
return NULL;
```

Type 2: string literal different

Type 1: white space different

Type 1 & 2 clones

```
const char *err = ap_check_cmd_context(cmd, GLOBAL_ONLY);
if (err != NULL) {
    return err;
}
ap_threads_per_child = atoi(arg);
if (ap_threads_per_child > thread_limit) {
    ap_log_error(APLOG_MARK, APLOG_STARTUP, 0, NULL,
                 "WARNING: ThreadsPerChild of %d exceeds ThreadLimit "
                 "value of %d threads,", ap_threads_per_child,
                 thread_limit);
    ....
    ap_threads_per_child = thread_limit;
}
else if (ap_threads_per_child < 1) {
        ap_log_error(APLOG_MARK, APLOG_STARTUP, 0, NULL,
                 "WARNING: Require ThreadsPerChild > 0, setting to 1");
        ap_threads_per_child = 1;
}
return NULL;
```

Type 2 clones

```
static GnmValue *
gnumeric_oct2bin (FunctionEvalInfo *ei, GnmValue const * const *argv)
{
   return val_to_base (ei, argv[0], argv[1],
        8, 2,
        0, GNM_const(7777777777.0),
        V2B_STRINGS_MAXLEN | V2B_STRINGS_BLANK_ZERO);
}

static GnmValue *
gnumeric_hex2bin (FunctionEvalInfo *ei, GnmValue const * const *argv)
{
   return val_to_base (ei, argv[0], argv[1],
        16, 2,
        0, GNM_const(9999999999.0),
        V2B_STRINGS_MAXLEN | V2B_STRINGS_BLANK_ZERO);
}
```

Type 2: numeric literal different

Type 2: identifier different

Type 3 clone

```
static PyObject *
py_new_RangeRef_object (const GnmRangeRef *range_ref){
   py_RangeRef_object *self;
   self = PyObject_NEW py_RangeRef_object,
        &py_RangeRef_object_type);
   if (self == NULL) {
        return NULL;
   }
   self->range_ref = *range_ref;
   return (PyObject *) self;
}
```

Type 3 clone

```
static PyObject *
py_new_Range_object (GnmRange const *range) {
  py_Range_object *self;
  self = PyObject_NEW (py_Range_object,
      &py_Range_object_type);
  if (self == NULL) {
      return NULL;
  }
  self->range = *range;
  return (PyObject *) self;
}
```

Type 3 clone

Type 3: different tokens (order)

```
static PyObject *
py_new_Range_object (GnmRange const *range) {
  py_Range_object *self;
  self = PyObject_NEW (py_Range_object,
      &py_Range_object_type);
  if (self == NULL) {
      return NULL;
  }
  self->range = *range;
  return (PyObject *) self;
}
```

Type 2: identifier different

A more common type 3 clone

```
static PyObject *
py_new_Range_object (GnmRange const *range) {
  if (!DEBUG) {
      py_Range_object *self;
      self = PyObject_NEW (py_Range_object,
            &py_Range_object_type);
      if (self == NULL) {
            return NULL;
      }
  } else {
      return NULL;
  }
  self->range = *range;
  return (PyObject *) self;
}
```

Measuring detection effectiveness

- We borrow these terms from IR:
  - Precision: How many of the answers you find are real?
  - Recall: How many of the real answers do you find?

  … but we usually lack “ground truth”

- False positives and filtering:
  - Most detection tools are highly tunable
  - Often set tool for “more hits”, then perform customized filtering to remove common false positives

More of the same, only different

- Problems related to software clone detection
  - Plagiarism detection, IP theft, GPL violations
  - DNA sequence analysis
  - Data compression
  - SPAM analysis, malware detection

Code clone detection methods

Structural

- Sequences
  - Strings
  - Tokens
- Graphs
  - ASTs
  - PDGs

Time, complexity, prog lang dependence

Others / hybrids

- Metrics
- Lightweight semantics
- Normalization
- Analyzing assembler

See also Roy & Cordy's tech report

Sequence-based approaches

- Atomic unit of comparison?
  - … could be char string (LOC), token, assembler instruction, SHA1 hash code, …
  - Comparison between atoms could be exact or approximate
- To find clones of sequences of atoms:
  - Longest exact sequential match
    - Use suffix tree/array
  - Compute Levenstein edit distance
  - Use n-grams and a moving window to allow for gaps [Baker, Johnson, MOSS]

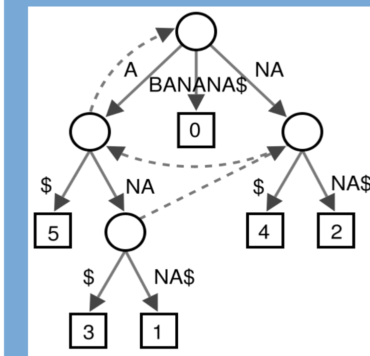
String-based clone detection

- Model:
  - Programs are *sequences* of *character strings* (i.e., LOC)
- Simple to implement
  - Look for exact textual matches of LOC
  - Mostly independent of prog lang
- Typical:
  - Pre-process to remove white space + comments
  - Convert strings to hashes to speed comparison
- Variants:
  - Allow gaps in sequences
  - Use Levenstein edit distance for near-misses

Token-based clone detection

- Model:
  - Programs are *sequences* of *tokens*
- Low dependence on program lang!
- Typical:
  - Use suffix trees/arrays to detect results

Suffix tree / array



[Diagram from Wikipedia]

- For each token stream ("string"), build a tree that represents all possible suffixes
  - Compare each new string to the set of existing trees
  - Comparisons are fast, but uses a lot of space
- Suffix array is a *sorted* list of all suffixes:
  1. a
  2. ana
  3. anana
  4. banana
  5. na
  6. nana
- Constructing a suffix array is O(N)

Token-based clone detection

- Variants
  - Naïve generalized token stream
    - Map ids to <ID>, string values to <STRING>, etc
    - So x = x + 1 becomes <id> = <id> + <NUM>, and will match these:

      y = y + 1
      a = b + 5
      x = y + 3.14

      but not these:

      x = 1 + x
      x++
      x = x + y

Token-based clone detection

- Variants
  - Smarter generalized token streams
    - Can add back in structural (unification) constraints

      e.g., require that if we are matching x = x + 1, then insist that any match have the same variable name for tokens 1 and 3
    - Many tools don't bother, as it slows down analysis.
      - We can afford to be a bit liberal in matching, as we are typically looking for many matching tokens in a row (say 50) before we consider the fragments to be clones

Token-based clone detection

- Variants
  - External tools “mark up” entity boundaries of token stream
    - Break potential clones at these boundaries
    - We used `ctags` to make sense of CC-Finder output
  - Add knowledge of prog lang to improve results
    e.g., `switch` stmts cause many false positives in C/C++/Java
  - Use assembler instructions instead of source code tokens

Graph-based approaches

- Many of the same issues as sequence-based approaches, but the underlying structure is a graph or tree
  - This means that naïve matching is much more expensive
  - … so most use hashes, or similar heuristics

AST-based clone detection

- Model:
  - Generate, then compare abstract syntax trees / graphs
    - Hijack an existing compiler
- Computationally expensive, but also very accurate
  - Usually, be selective about when to “go deep”
- Obviously, it’s prog lang dependent

AST-based clone detection

- Variants:
  - Use parse trees instead
  - Use suffix trees to store ASTs [Koschke]
  - Walk trees to generate metrics instead of comparing trees structurally [Kontogiannis]
  - Combined metrics / AST [Deckard]
    - At each node in parse tree, store feature vector of keywords that occur in subtree
    - Take Euclidean distance of feature vector to compare potential clones

PDG-based clone detection

- Model:
  - Program dependence graphs (PDGs) are used in *program slicing* to compute the sub-programs that concern only particular variables / statements
- Can trace through / compare “paths of interest”
  - Easy to ignore gaps / interruptions in code
  - Largely immune to simple re-orderings of lines
- Naïve version is very expensive
  - So need to be clever about when + how to do it

Metrics-based detection

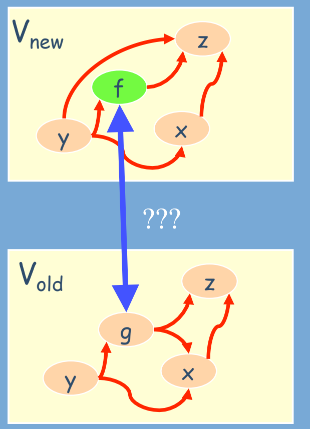
- Compare measurements instead of structures
  - Hashes
  - LOC, McCabe, fan-in/out
  - # methods/fields, signatures only
- Main advantage: speed
  - Do expensive processing in one pass
  - … then compare numbers / vectors pairwise
- Also, metrics are largely immune to simple re-orderings

Metrics-based detection

- Just about any of the previous methods can use metrics judiciously
  - Hash complex structures to simple values
  - Within buckets, selectively perform more expensive comparisons

Lightweight semantics

- Origin analysis [Godfrey/Zou 05]
  - Given consecutive versions of a system, which “new” entities really are new, and which are “old” entities that have been moved, renamed, merged, split, refactored
- Perform via
  - Entity analysis (trad. clone detection)
  - Relationship analysis (e.g., call sets)
  - Known patterns of change



Source normalization

- Can be performed in addition to other approaches, typically as a preprocessing step
- Examples:
  - Remove comments and/or whitespace
    - Alternatively, could look *only* at comments! (ignoring boilerplating, GPL, annotations)
  - Run source through pretty printer
  - Use source transformation tool [Roy/Cordy]
    - Map **`while/for/do`** and **`if/switch`** to single canonical representations
    - Ignore structures that we don't care about
  - Compile sources and perform clone detection on binaries!
    - Compilers are very good at source normalization

So … what's the best approach?

- It's a very hard question to answer
  - We don't agree on what a clone is
  - Different tools produce very different results
    - They're probably all clones, depending on your definition ☺
  - There is a benchmark dataset [Bellon/Koschke], but it's only a first step and it needs updating
  - Some comparative studies have been done, but the results are mixed [Bailey, Bellon, van Rysselberghe]
- At best, we can say we understand some of the trade-offs between the various approaches
  - Efficiency, accuracy per clone type, etc.

Clone detection summary

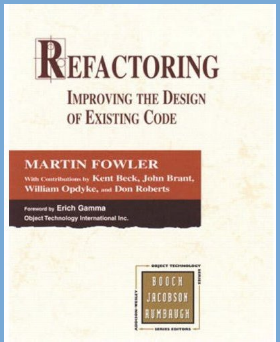
- Approaches vary in complexity, programming language dependence, applicability
- Combining shallow + deep analysis judiciously seems key
  - **Cheap:** Comparing hashes, metrics values, feature vectors
  - **Expensive:** Computing hashes, graph/tree walking, dynamic programming
  - Many SE researchers use a tweaked token-based and/or suffix tree implementations
- Unclear how to measure progress, compare results

Cloning and software engineering

Just how bad is software cloning?

- Most early research concentrated on *detection* algorithms
- Recent focus has shifted to clone *analysis*
  - *What techniques are effective to study large systems?*
  - *What kinds of clones are there? What properties do they have?*
  - *Does cloning really hurt the design in the long run?*
- Case studies suggest that cloning is common practice in industrial software
  - 5-15% is common; up to 50% in some systems
  - … but it's unclear how "bad" this is

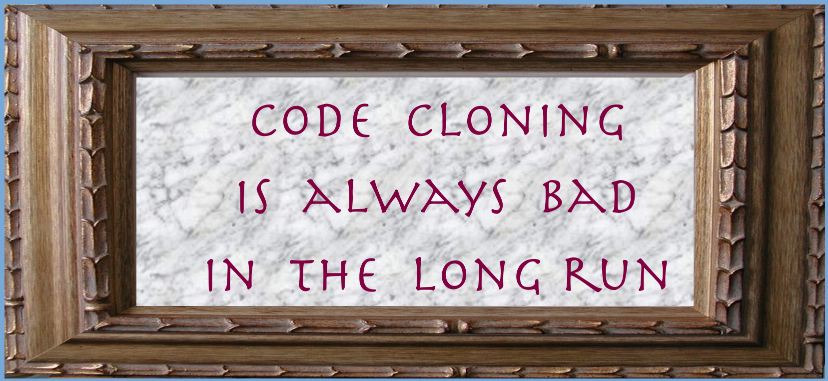
Quotes on source code cloning



"Number one in the stink parade is duplicated code. If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them."

– "Bad Smells" [Beck/Fowler in *Refactoring*]

Myth



CW: *Cloning is bad because …*

- Sloppy design, lazy developers, incurred technical debt leads to bloated, crufty designs
  - Cruft *accumulates* and *ossifies*
  - Can't remove it, can't understand it, need to work around it
- Inconsistent maintenance
  - Did you fix / adapt all of the clones too?

What to do? Refactor, refactor, refactor!

- Move common stuff to parent class, use generics, parameters, …

What you are supposed to do instead

1. Identify commonalities across code base
2. Refactor duplicate functionality to one place in the code
   - Functions with parameters
   - Base class encapsulates commonalities, derived classes specialize peculiarities
   - Generics / templates for classes / functions / (aspects?)

Cloning is bad?

[from Handel's Messiah]

All we like sheep
All we like sheep
All we like sheep
All we like sheep
… have gone astraaaaaay

OK, but there's no need for repetition in …

- Engineering!
  - Well … engineering often achieves scalability by repeating trusted design elements
- Software engineering!
  - Umm … server farms, virtual machines, map-reduce
- Software design!
  - Errr … XP's Rule of 3, the Prototype design pattern, COBOL boilerplating, design cookbooks

Clone analysis (and not detection)

Some empirical results

Clone genealogies [Kim et al. 2005]

Q: How and why do clones change over time?

- Looked at two ~20 KLOC Java programs (CAROL+ dnsjava)
- Findings:
  - Some clones are "volatile", are introduced as a means-to-an-end but get refactored and disappear quickly
  - Some clones are more long lived, often hard to refactor due to programming language limitations (e.g., lack of generics, aspects)
  - Many clones are maintained in parallel, but some are not
  - It's common for clones to change in different ways over time
  - Some clones represent fundamental design decisions that can't be refactored easily
  - Naïve aggressive refactoring is not the answer!

Consistency of change [Krinke 2007]

Q: Is inconsistent maintenance of clones really a problem?

- Studied five large open source systems over time:
  - ArgoUML, CAROL, jdt.core, emacs, FileZilla
- Findings:
  - Clone groups are changed consistently about 50% of the time
  - Clone groups that are not changed consistently rarely become so later
    - So probably they were intended to diverge

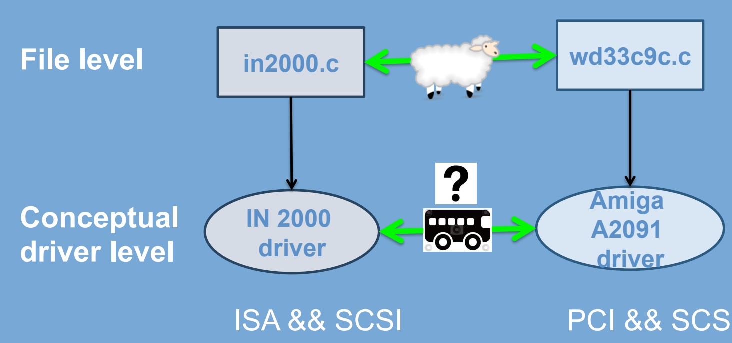
Clones: What is that smell? [Rahman et al. 2010]

Q: Is cloned code buggier than non-cloned code?

- Examined several large open source projects:
  - Evolution, Apache, Gimp, Nautilus
- Findings:
  - Most bugs have very little to do with clones,
    - Cloned code is typically *less* buggy than non-cloned code
  - Larger clone groups *don't* have more bugs than smaller groups
    - Making more copies of code *doesn't* introduce more defects,
    - Larger clone groups (# of members) have *lower* bug density per line than smaller clone groups.

Linux SCSI driver cloning [Wang/Godfrey 2011]

Q: Does cloning predict compatible bus type dependencies?



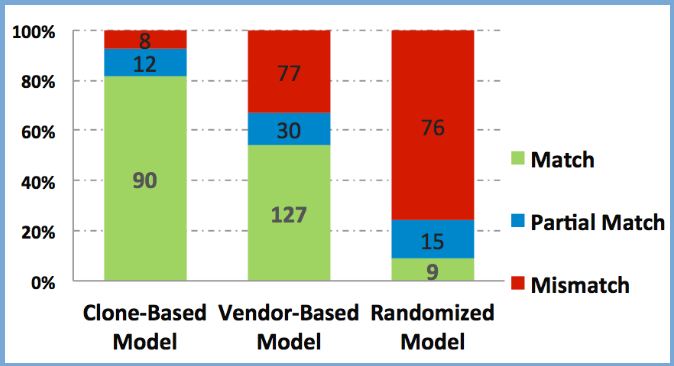
Linux SCSI driver cloning

Matching bus type dependencies:

1. Extract dependency info from config files
2. Convert each logical expression into DNF
3. Run matcher

	(ISA && SCSI) || (PCI && SCSI)	ISA && SCSI && PCI	SCSI && X86_32
ISA && SCSI	*Match*	*Partial match*	*Mismatch*

Predictive power of cloning



Clone analysis beats domain knowledge!

Cloning considered harmful?

A taxonomy of cloning intent

1. Forking
   - Hardware variation
     e.g., Linux SCSI drivers
   - Platform variation
   - Experimental variation
2. Templating
   - Boilerplating
   - API / library protocols
   - Generalized programming idioms
   - Parameterized code
3. Post-hoc customizing
   - Bug workarounds
   - Replicate + specialize

["'Cloning considered harmful' considered harmful: Patterns of cloning in software", Cory J. Kapser and Michael W. Godfrey, *Empirical Software Engineering,* 2008]

1. Forking

- Often used to "springboard" new or experimental development
  - Clones will need to evolve independently
  - Big chunks are copied!
- Works well when the commonalities and differences of the end solutions are unclear

1. Forking: Platform variation

- Motivation
  - Different platforms ⇒ very different low-level details
  - Interleaving platform-specific code in one place is too complex
- Well-known examples
  - Linux kernel "arch" subsystem
  - Apache Portable Runtime (APR)
    - Portable impl of functionality that is typically platform dependent, such as file and network access
    - `fileio -> {netware, os2, unix, win32}`
    - Typical diffs: insertion of extra error checking or API calls
    - Cloning is obvious and well documented

1. Forking: Platform variation

- Advantages of cloning
  - Each (cloned) variant is simpler to maintain
  - No risk to stability of other variants
  - Platforms are likely to evolve independently, so maintenance is likely to be "mostly independent"
- Disadvantages of cloning
  - Evolution in two dimensions: user reqs + platform support
  - Change to the interface level means changes to many files

1. Forking: Platform variation

- Management and long-term issues
  - Factor out platform independent functionality as much as possible
  - Document variation points + platform peculiarities
  - As # of platforms grows, interface to the system hardens
- Structural manifestations
  - Cloning usually happens at the file level.
    - Clones are often stored as files (or dirs) in the same source directory
    - Dirs may be named after OSs or similar

2. Templating

- Code embodying the desired behavior already exists
  - … but the impl. language does not provide strong support for the desired abstraction
- Linked editing or source auto-generation can be used
- Examples
  - COBOL boilerplate code
  - C routines that treat floats and ints analogously
  - (old) Java code that could have used generics
  - API usages for common tasks (eg GUI creation)
  - Language / platform idioms, such as safe pointer handling

3. Post-hoc customizing

- Existing code solves a similar problem but you can't or won't change it
  - May not own the code [Microsoft: "Clone and own"]
  - May not want to risk change there
    - Changing may be complex, safer to play elsewhere
- Examples:
  - Replicate and specialize
  - Bug workarounds

Cloning harmfulness:
Two open source case studies

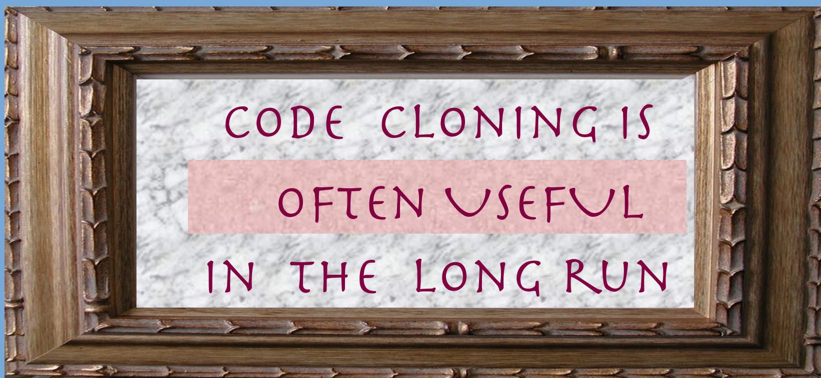
		Apache		Gnumeric	
Group	**Pattern**	**Good**	**Harmful**	**Good**	**Harmful**
Forking	Hardware variation	0	0	0	0
Forking	Platform variation	10	0	0	0
Forking	Experimental variation	4	0	0	0
Templating	Boiler-plating	5	0	6	7
Templating	API	0	0	0	9
Templating	Idioms	0	12	1	1
Templating	Parameterized code	5	12	10	34
Customizing	Replicate + specialize	12	4	15	16
Customizing	Bug workarounds	0	0	0	0
Total		**36**	**28**	**32**	**67**

Apache httpd 2.2.4 - 60 Tokens
Gnumeric 1.6.3 - 60 Tokens

["'Cloning considered harmful' considered harmful: Patterns of cloning in software", Cory J. Kapser and Michael W. Godfrey, *Empirical Software Engineering,* 2008]

Myth

~~Myth~~ Motto



Summary

- Cloning is pretty common in industrial code!
  - Often it's done in a principled way …
  - … so refactoring may be a *bad* idea
  - … so we need to consider context + rationale before refactoring
- Empirical evidence from open source systems suggests:
  - There are many reasons to clone
  - Cloned code is often maintained appropriately
  - Principled cloning doesn't seem to cause undue problems later on … so it was probably the right design choice!

All we like sheep:
Cloning as a software engineering tool

Michael W. Godfrey
University of Waterloo