CS6201 Software Reuse

CS6201 Lecture Notes: Set #3

Topic: Software clones and generic design

1. What are software clones?
2. Clones in STL
3. A study of cloning in the Buffer library
4. Buffer library in Java/XVCL representation
5. Cloning phenomenon and research on clones
6. Clone detection

--- relevant papers in references

Software clones

- Repetitions; similar program structures, recurring in the same or similar form

1. simple clones - the same or similar code fragments
   - similar functions, class methods, any code fragments
2. structural clones – design-level similarities
   - any similar program structures
   - similar classes, components
   - patterns of collaborating components

Types of simple clones

```
int foo(byte) {
int c;
   While (isalpha(c)) {
        if (p == token_buffer)
        p = grow_token_buffer(p);
        c = getc(finput); }
}
```

Exact clones

```
int foo(byte) {
int c;
   While (isalpha(c)) {
        if (p == token_buffer)
        p =
     grow_token_buffer(p);
        c = getc(finput); }
}
```

Parametric clones

```
char bar(long)  {
char d;
   While (isdigit(d)) {
        if (p == token_buffer)
        p = grow_token_buffer(p);
      d = getc(finput); }
}
```

Gapped clones

```
char foo(long, float)  {
char d;
   While (isdigit(d)) {
        if (p == token_buffer)
        p =
     grow_token_buffer(p);
        if (d == '-') return;
        d = getc(finput); }
}
```


Structural clones

any similar program structures:

- similarity patterns in requirements
  - recurring analysis problems
- similar design-level structures
  - repeatedly used design solutions (patterns)
  - design patterns, enterprise patterns (J2EE, .NET)
    - patterns of collaborating classes
  - architectural patterns
  - mental templates used by programmers

design of generic program solutions to unify structural clones is an important theme of software reuse

Similar classes

```
class X {
  f ()
  g ()
  h ()  }
```

```
class Y {
  g ()
  h ()
  f ()  }
```

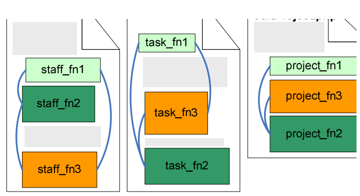
```
class A {
  int foo (byte)
    { a b c d }
  g ()    }
```

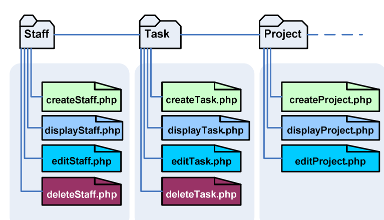
```
class B {
  char bar (long)
    { b d c }
  // missing g () }
```

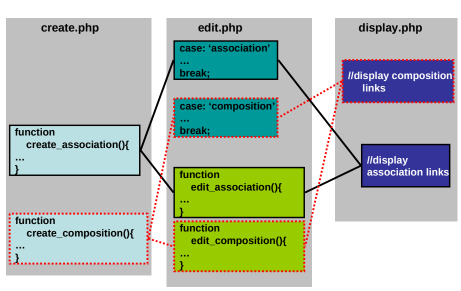
```
class C {
  char foo (long, float)
    { a c d }
  h () extra method }
```

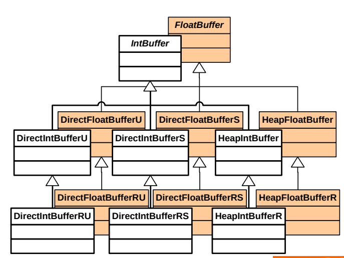

Other structural clones

Large-granularity repetitions, configurations of components

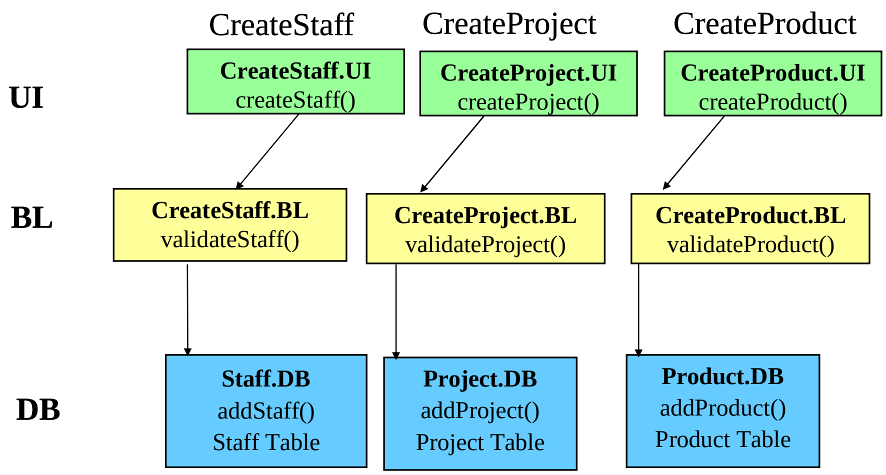








Group of similar operations: Create[E]

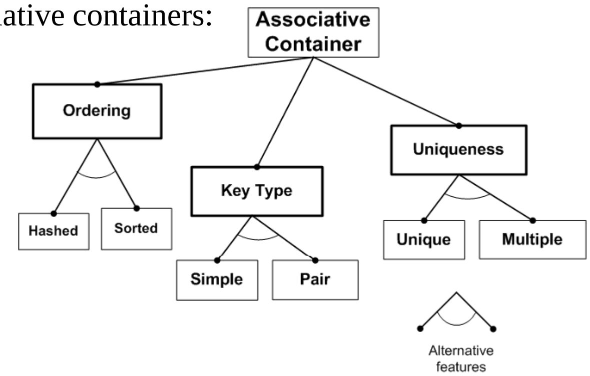


Clones in STL?

details of this case study are described in Basit, H.A., Rajapakse, D.C., and Jarzabek, S. "Beyond Templates: a Study of Clones in the STL and Some General Implications," *Int. Conf. Software Engineering, ICSE'05*, St. Louis, USA, May 2005, pp. 451-459

Associative containers

- variable-sized containers supporting efficient retrieval of elements via keys
- features in associative containers:



- eight templates implementing eight feature combinations

Clone examples

```
template <class _Key, class _Compare, class _Alloc>
inline bool operator == (
    const set<_Key,_Compare,_Alloc>& __x,
     const set<_Key,_Compare,_Alloc>& __y) {
 return __x._M_t ==__y._M_t;
}
-----------------------------------------------------------------------------------------
template <class _Key, class _Compare, class _Alloc>
inline bool operator < (
    const set<_Key,_Compare,_Alloc>& __x,
     const set<_Key,_Compare,_Alloc>& __y) {
 return __x._M_t < __y._M_t;
}
```

Clone examples, cont.

```
template <class _Tp>
inline valarray<_Tp> operator+(
  const valarray<_Tp>& __x, const _Tp& __c) {
 typedef typename valarray<_Tp>::_NoInit _NoInit;
 valarray<_Tp> __tmp(__x.size(), _NoInit());
 for (size_t __i = 0; __i < __x.size(); ++__i)
   __tmp[__i] = __x[__i]  + __c;
 return __tmp;
}
----------------------------------------------------------------------------------------------------
template <class _Tp>
inline valarray<_Tp> operator+(
 const _Tp& __c, const valarray<_Tp>& __x) {
 typedef typename valarray<_Tp>::_NoInit _NoInit;
 valarray<_Tp> __tmp(__x.size(), _NoInit());
 for (size_t __i = 0; __i < __x.size(); ++__i)
   __tmp[__i] = __c + __x[__i];
 return __tmp;
}
```


Cloning level in STL

templates	cloned code
8 assoc. containers	50% of cloned code, *can be unified by 2 power-generics*
stack and queue	40% of cloned code
set operations: union, intersection, etc.	50% of cloned code, can be unified by one generic operation
iterators	no clones found

Clones in Buffer Library

JDK 1.4.1 and 1.5

basic study described in: Jarzabek, S. and Li, S. "Eliminating Redundancies with a "Composition with Adaptation" Meta-programming Technique," *Proc. ESEC-FSE'03, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press, September 2003, Helsinki, pp. 237-246.

Jarzabek, S. and Li, S. "Unifying Clones at the Meta-Level for Enhanced Changeability: A Case Study and General Implications," This is an extended version of the Buffer library experiment, described in [1]. Analysis of OO shortcomings in addressing redundancies is more comprehensive and accurate in this version than in the short version [1].

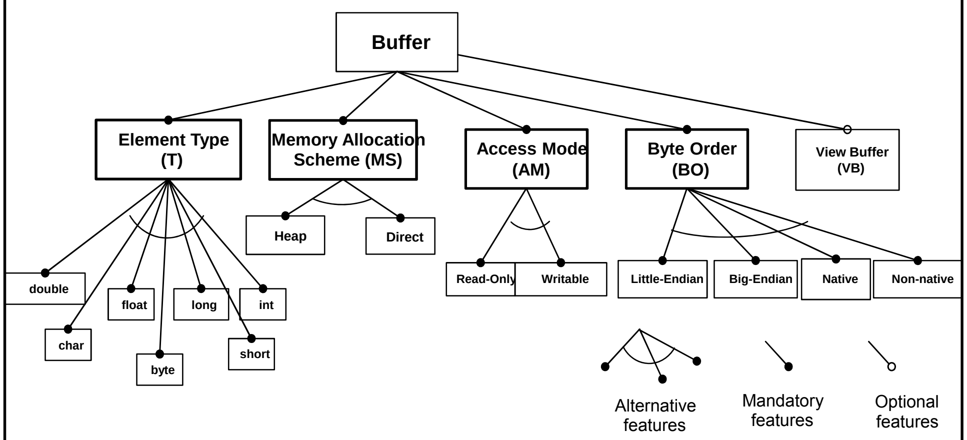
Jarzabek, S. "A Case for Enhancing Generic Design at the Meta-Level: Motivation, an Example, a Method and Its Evaluation," better description of similarity pattern problem, taking into account a number of empirical studies.

Buffer library

- buffer contains data in a linear sequence for reading and writing
- buffers differ in features such as:

buffer element type	byte, char, int, float, double, long, short
memory allocation scheme	Direct, Heap
byte ordering	Native, Non-native, Big_endian, Little_endian
access mode	Writable, Read-only

Features in the Buffer library



Each combination of features represents a unique buffer → 74 classes

Buffer classes

- each combination of features yields a unique class

[MS][T]Buffer[AM][BO]

MS – memory scheme: Direct, Heap

T – type: Byte, Char, Int, Double, Float, Long, Short

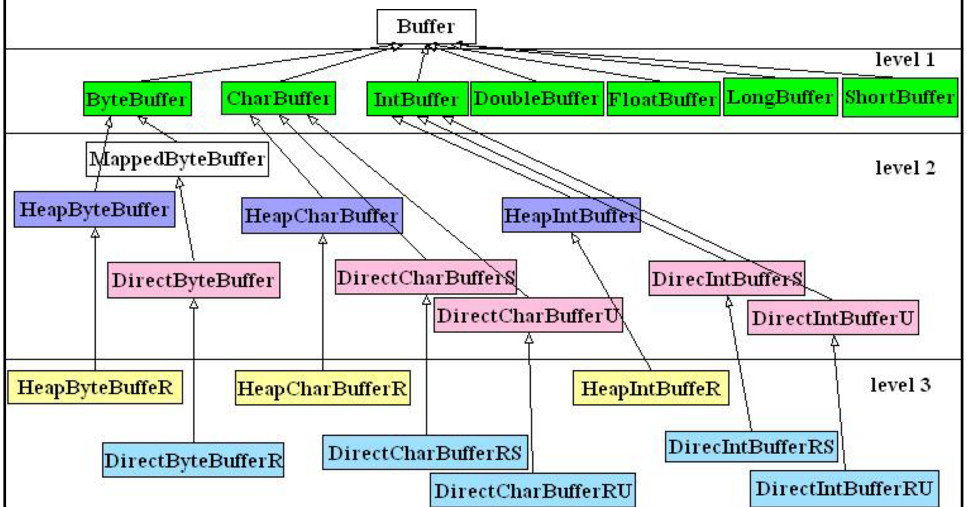
AM – access mode: R - read-only, W - writable

BO – byte ordering: S – non-native U – native, B – BigEndian, L - LittleEndian

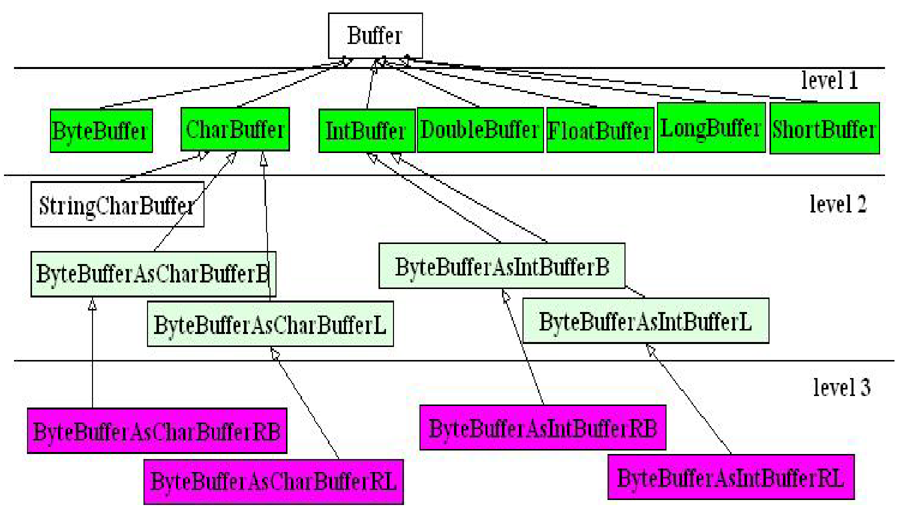
- e.g., class DirectIntBufferRS

MS = Direct, T = Int, AM = R, BO = S

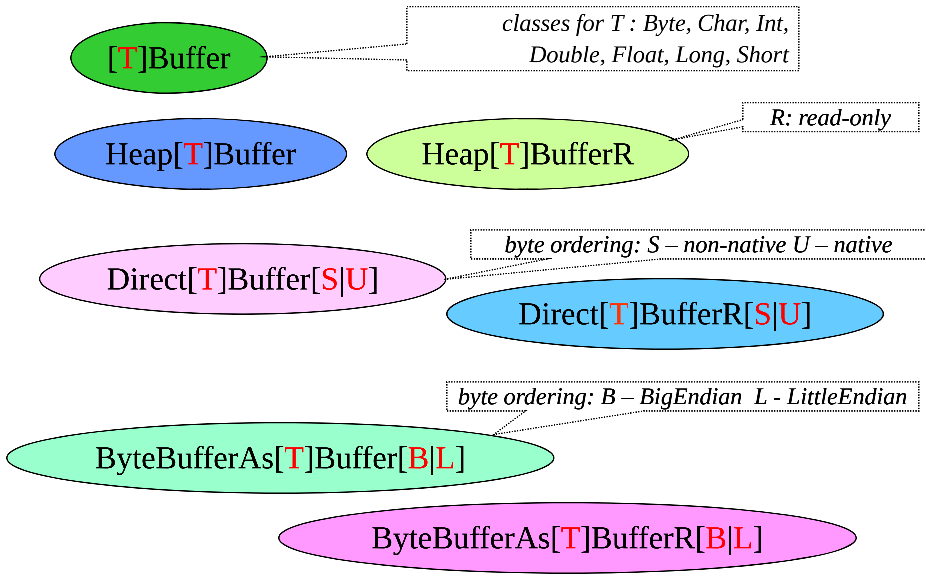
Buffer library



Buffer library

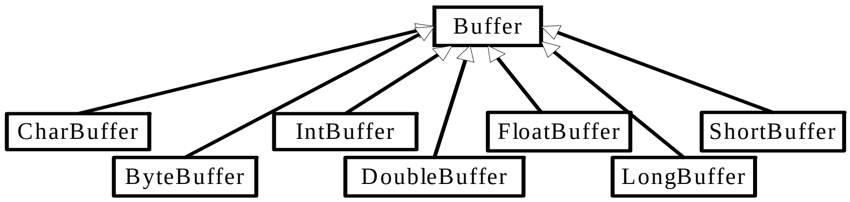


Groups of similar classes



Can we refactor classes with inheritance or generics to avoid cloning?

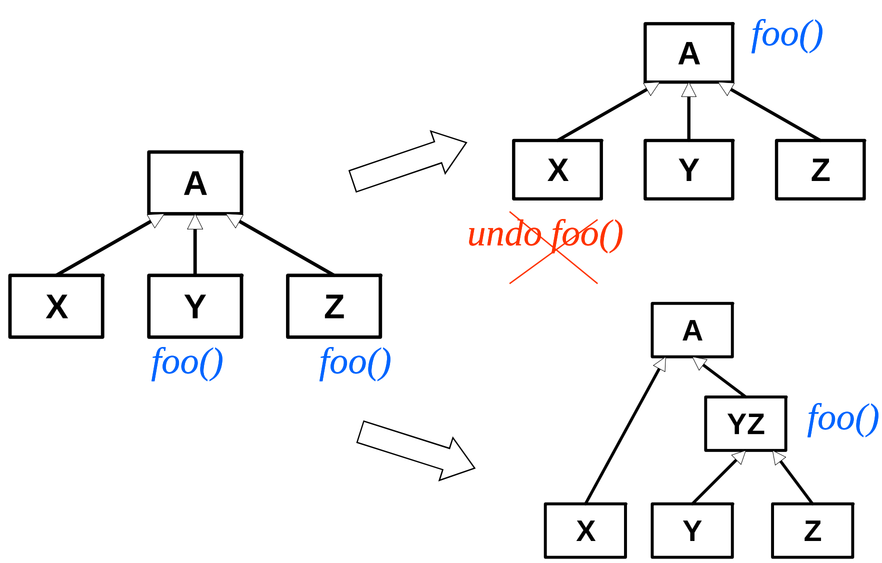
Method *hasArray() repeated in 7 classes*



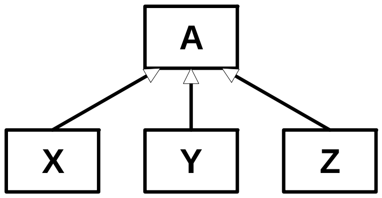

```
class CharBuffer {
    char  hb;
    public final boolean hasArray () {
    return (hb != null)  && !isReadOnly ; } }
```

```
class IntBuffer {
    int  hb;
    public final boolean hasArray () {
    return (hb != null)  && !isReadOnly ; } }
```


Inheritance : example 1



Inheritance: example 2



foo(int) *foo()* *foo()*

int foo()

foo()

```
class X {
  int x;
  foo () {
        x = 2;  }
}

class Y {
  double y;
  foo () {
        y = 5;  }
}
```


Java 1.5 with generics

- generic type parameters to be replaced by concrete types [T] Stack
- an ideal solution: a generic Buffer:

  [T, AM, MS, BO, VB] Buffer

T – type; MS – memory scheme, AM – access mode,

- unrealistic – the impact of features on code cannot be easily parameterized

Generics-friendly buffer classes

- 15 classes that differ in types only:
  - [T] Buffer, [T] HeapBuffer and [T] HeapBufferR
  - types T: int, short, long, float, double
- unification of 15 generics-friendly classes with 3 generic classes saves 27% of code
- limitations of Java generics:
  - primitive types: int, short, … not allowed as arguments
  - no generic methods (only classes)

Generic classes (templates)

- can unify classes that differ in type parameters
  - uniform propagation of type parameters across classes
- cannot unify other differences:
  - **class A: a + 5** **class B: a - 7**
  - other algorithmic details, added/deleted fragments
  - ad hoc combinations of differences across similar methods
- couplings among classes impede application of generics

Differences among cloned classes in each group

- type parameters in attribute declarations and methods
- non-type parameters
  - operators, keywords, constants, names
- minor or major editing changes
- different implementation of the same method
- extra methods in certain classes
- details in method signatures, ‘implements’ clause, etc.

An example of method clone

```
/*Creates a new byte buffer */
public ByteBuffer slice() {
        int pos = this.position();
        int lim = this.limit();
        assert (pos <= lim);
        int rem = (pos <= lim ? lim - pos : 0);
        int off = (pos << 0);
        return new DirectByteBuffer (this, -1, 0, rem, rem, off);
}
```

Generics-unfriendly variations

- non-type parametric differences:
  - constants, operators, keywords, names – any text fragment

```
private int doSomething(Integer op1, Integer op2) {
      Integer retval = doSomethingElse(op1,op2);
      print(retval);
      return retlval;
}
```

```
protected int doSomething(Double op1, Double op2) {
      Double retval = doSomethingElse(op1,op2);
      print(retval);
      return retlval;
}
```


Generics-unfriendly variations

```
public abstract class CharBuffer
    extends Buffer implements Comparable, CharSequence{
```

```
public String toString() {
  StringBuffer sb = new StringBuffer();
  sb.append(getClass().getName());
            ...
  sb.append(capacity());
  sb.append("]");
  return sb.toString();      }
```

```
    public String toString() {
       return toString(position(), limit());}
```

- extra methods in some of the classes

Couplings among classes

- couplings may restrict the use of generics

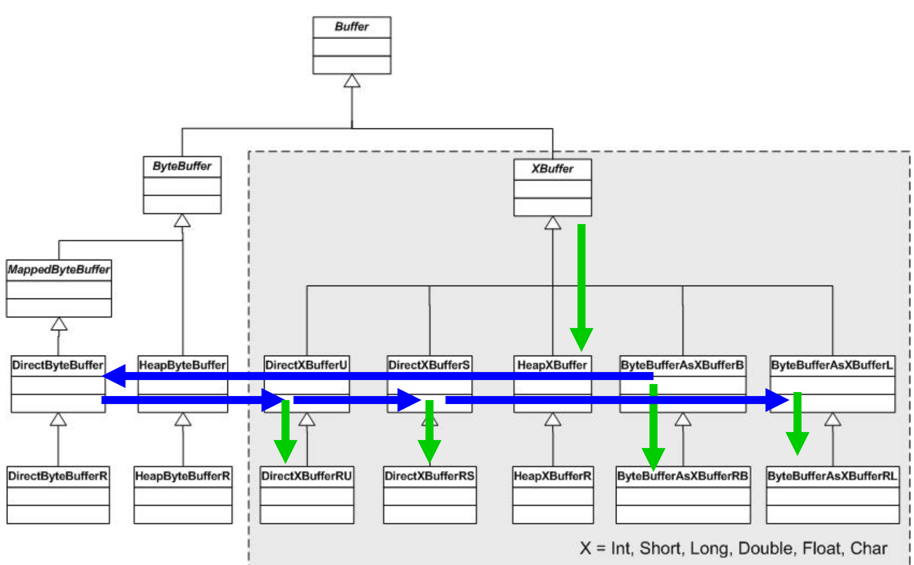
method get(int) in class DirectIntBufferS

```
public int get(int i) {
  return Bits.swap(unsafe.getInt(ix(checkIndex(i)));}
```

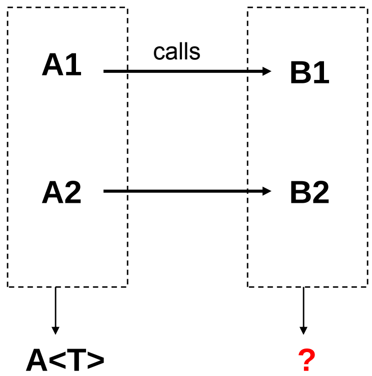
method get(int) in class DirectFloatBufferS:

```
public float get(int i) {
  return Bits.swap(unsafe.getFloat(ix(checkIndex(i)));}
```

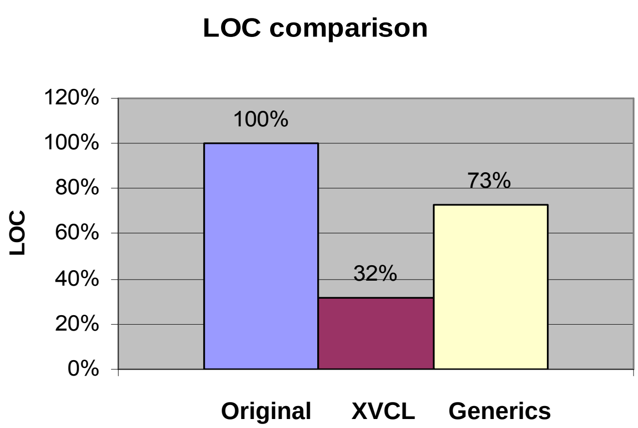

Couplings among classes



Couplings



The extent of cloning



Roots of cloning

1. "Feature combination" problem
   - Features cannot be implemented as separate modules
   - Interactions among features are chaotic
2. Multiple, competing design goals:

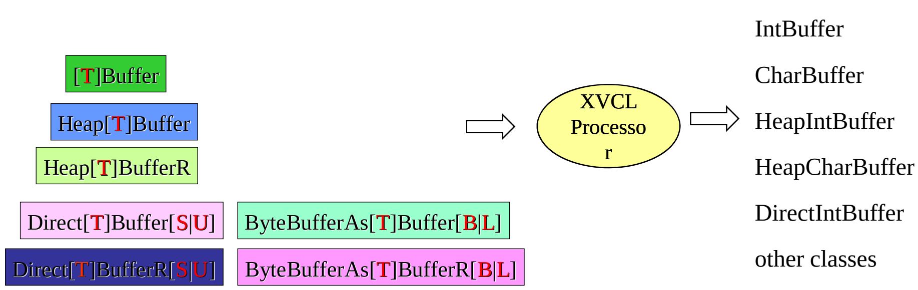
- usability, conceptual clarity
  - 1-to-1 mappings between concepts and classes, one class per combination of features
  - only top eight classes revealed to programmers
- performance, reliability, non-redundancy

When we can't avoid clones

- Some clones are unavoidable
  - Programming language limitations
  - Clones for performance or reliability reasons
  - Standardization, pattern driven design (.NET, JEE)
- Some clones can be avoided in theory, but a non-redundant solution is:
  - Complex, creates more problems than it solves
  - Compromises yet other important design goals
- Risk involved in refactoring clones (Cordy)

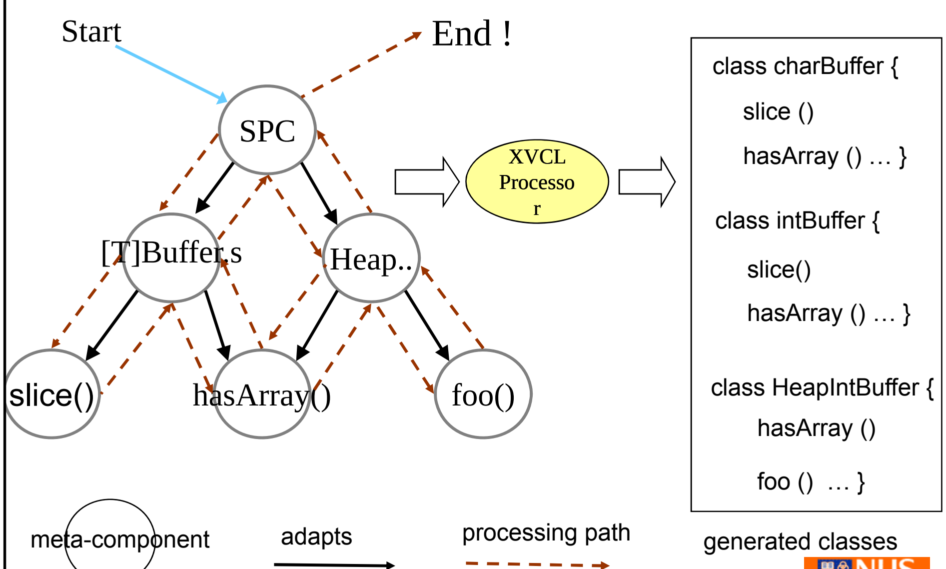
Buffer library in Java/XVCL

The concept of Buffer library in Java/XVCL



- Each group of similar buffer classes represented as generic, adaptable Java/XVCL meta-component (x-frame)
- Buffer classes are generated from Java/XVCL representation
- Java/XVCL used for maintenance and reuse

XVCL Processor at work



[T]Buffer group of classes

- five numeric buffer element types: int, short, etc.

```
public abstract class IntBuffer extends Buffer
extendsBuffer implements Comparable
{  final int[] hb;
   IntBuffer(int mark, int pos, int lim, int cap,
      int[] hb, int offset) {   … }
   IntBuffer(int mark, int pos, int lim, int cap)
   {… }
   public static IntBuffer allocate(int capacity)
   { …   return new HeapIntBuffer(capacity) }
   public static IntBuffer wrap(int[] array) { … }
   public abstract IntBuffer slice();
   public abstract IntBuffer duplicate();
    …
}
```

```
public abstract class ShortBuffer extends Buffer
extendsBuffer implements Comparable
{  final short[] hb;
   ShortBuffer(int mark, int pos, int lim, int cap,
      short[] hb, int offset) {   … }
   ShortBuffer(int mark, int pos, int lim, int cap)
   {… }
   public static ShortBuffer allocate(int capacity)
   { …   return new HeapShortBuffer(capacity) }
   public static ShortBuffer wrap(short[] array) {…}
   public abstract ShortBuffer slice();
   public abstract ShortBuffer duplicate();
    …
}
```

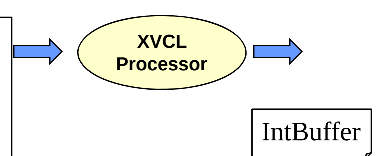
Java generics solution

```
public abstract class TBuffer extends Buffer <T>
extendsBuffer implements Comparable
{  final T[] hb;
   TBuffer(int mark, int pos, int lim, int cap,
       T[] hb, int offset) {   … }
   TBuffer(int mark, int pos, int lim, int cap)
   {… }
   public static TBuffer allocate(int capacity)
   { …   return new HeapTBuffer <T> (capacity) }
   public static TBuffer wrap(T[] array) { … }
   public abstract TBuffer slice();
   public abstract TBuffer duplicate();
    …
}
```


Generic [T]Buffer in Java/XVCL

```
<x-frame SPC >
<set Type = Int />
<set type = int />
<adapt [T]Buffer />
```

```
<x-frame [T]Buffer outfile  = @TypeBuffer.java >
public abstract class @TypeBuffer extends Buffer
extendsBuffer implements Comparable
{  final @type[] hb;
   Buffer(int mark, int pos, int lim, int cap,
       @type[] hb, int offset) {   … }
   @TypeBuffer(int mark, int pos, int lim, int cap)
   {… }
   public static @TypeBuffer allocate(int capacity)
   { …   return new Heap@TypeBuffer (capacity) }
      …
}
```



buffer class in Java

Generic [T]Buffer in Java/XVCL

```
<x-frame SPC >
<set Type = Short />
<set type = short />
<adapt [T]Buffer />
```

```
<adapt >
```

```
<x-frame [T]Buffer outfile  = @TypeBuffer.java >
public abstract class @TypeBuffer extends Buffer
extendsBuffer implements Comparable
{  final @type[] hb;
   Buffer(int mark, int pos, int lim, int cap,
        @type[] hb, int offset) {   … }
   @TypeBuffer(int mark, int pos, int lim, int cap)
   {… }
   public static @TypeBuffer allocate(int capacity)
   { …   return new Heap@TypeBuffer (capacity) }
      …
}
```

XVCL Processor

ShortBuffer

buffer class in Java

CS6201 Set #3 SP and GD © 2010 Stan Jarzabek NUS 43

Deriving five [T]Buffer classes

```
<x-frame SPC >
<set Type = Int, Short, Float, Long, Double />
<set type = int, short, float, long, double />
<while Type, type>
   <adapt [T]Buffer />
</while>
```

```
<adapt>ed five times
```

```
<x-frame [T]Buffer outfile  = @TypeBuffer.java >
public abstract class @TypeBuffer extends Buffer
extendsBuffer implements Comparable
{  final @type[] hb;
   Buffer(int mark, int pos, int lim, int cap,
        @type[] hb, int offset) {   … }
   @TypeBuffer(int mark, int pos, int lim, int cap)
   {… }
   public static @TypeBuffer allocate(int capacity)
   { …   return new Heap@TypeBuffer (capacity) }
      …
}
```

XVCL Processor

IntBuffer

ShortBuffer

FloatBuffer

LongBuffer

DoubleBuffer

buffer classes in Java

CS6201 Set #3 SP and GD © 2010 Stan Jarzabek NUS 44

Addressing class CharBuffer

differences between numeric buffer classes and CharBuffer

```
public abstract class IntBuffer extends Buffer
extendsBuffer implements Comparable
{  final int[] hb;
   IntBuffer(int mark, int pos, int lim, int cap,
       int[] hb, int offset) {   … }
    public String toString() { …  }
 }
```

```
public abstract class CharBuffer extends Buffer
extendsBuffer implements Comparable,CharSequence
{  final char[] hb;
   CharBuffer(int mark, int pos, int lim, int cap,
       char[] hb, int offset) {   … }
   public String toString() { different implementation }
   many extra methods in Char Buffer:
   public static CharBuffer wrap(CharSequence csq) { }
   etc.
 }
```

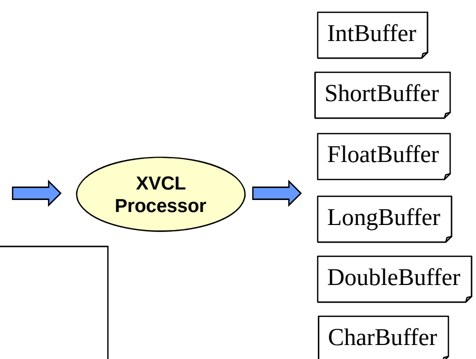
for CharBuffer we must:

1. update implements clause
2. re-define implementation of method toString(), and
3. insert extra methods


```
<x-frame SPC >
<set Type = Int, Short, Float, Long, Double, Char />
<set type = int, short, float, long, double, char />
<while Type, type>
  <select option = Type>
    <option Char>
       <adapt [T]Buffer />
         <insert implements >
           ,CharSequence
         <insert toString >
           implementation of method toString() for CharBuffer
         <insert extraMethods >
           implementation of extra methods for CharBuffer
   <otherwise>
      <adapt [T]Buffer />
</while>
```

<adapt>ed six times

```
<x-frame [T]Buffer outfile  = @TypeBuffer.java >
public abstract class @TypeBuffer extends Buffer
extendsBuffer implements Comparable <break implements>
{  final @type[] hb;
   Buffer(int mark, int pos, int lim, int cap,
      @type[] hb, int offset) {   … }
<break toString >
   implementation of method toString() for numeric classes
<break extraMethods >
   implementation of methods specific to CharBuffer
 }
```



buffer classes in Java

ByteBuffer has yet other extra methods …

Deriving seven [T]Buffer classes

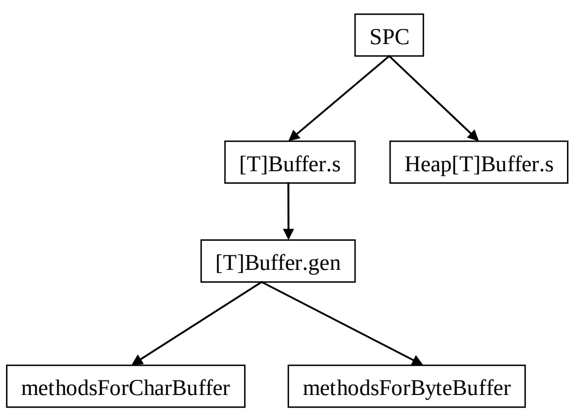
```
<x-frame SPC >
<set Type = Int, Short, Float, Long, Double, Char, Byte  />
<set type = int, short, float, long, double, char, byte />
<while Type>
    <select option = Type>
        <option Char>
            <adapt [T]Buffer>
                customizations for CharBuffer (as before)
        <option Byte>
            <adapt [T]Buffer>
                customizations for ByteBuffer (as before)
        <otherwise>
            <adapt [T]Buffer/>
    </select>
</while>
```

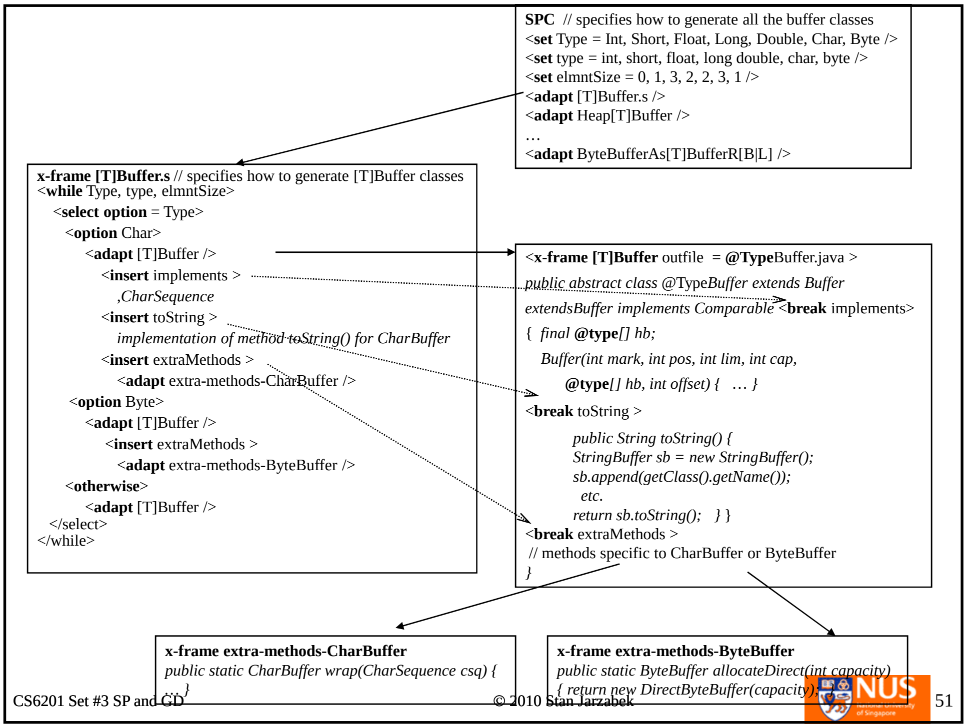
slice()

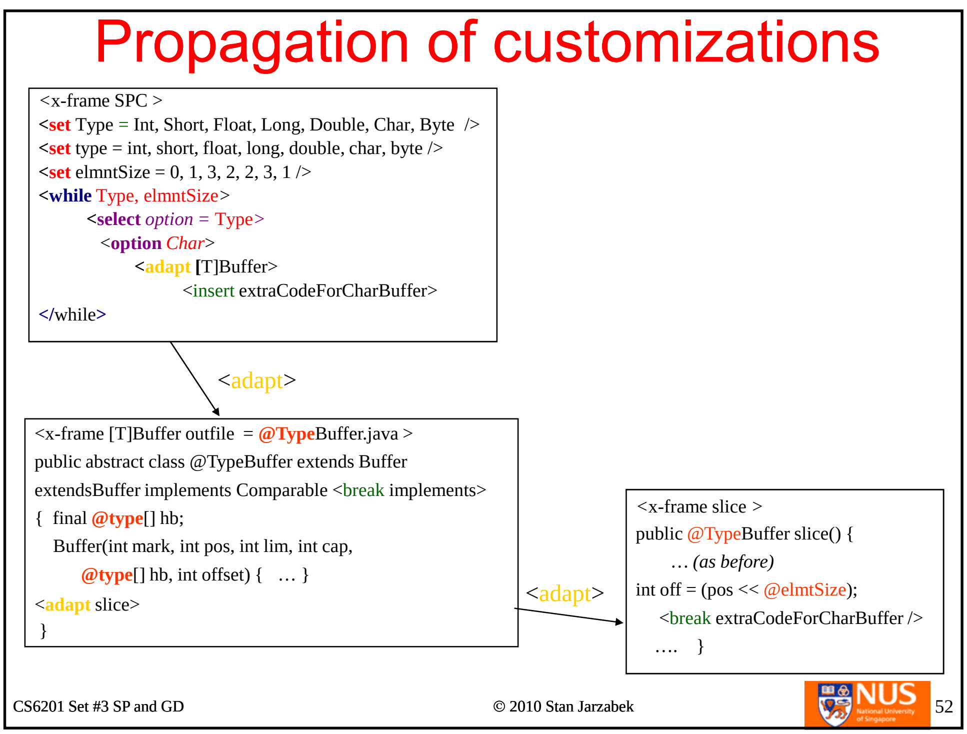
Generic method slice()

```
<x-frame slice >
public @TypeBuffer slice() {
   int pos = this.position();
   int lim = this.limit();
   assert (pos <= lim);
   int rem = (pos <= lim ? lim - pos : 0);
   int off = (pos << @elmtSize);
    <break extraCodeForCharBuffer />
    return new Direct@Type Buffer@byteOrder
    (this, -1, 0, rem, rem,  off); }
```

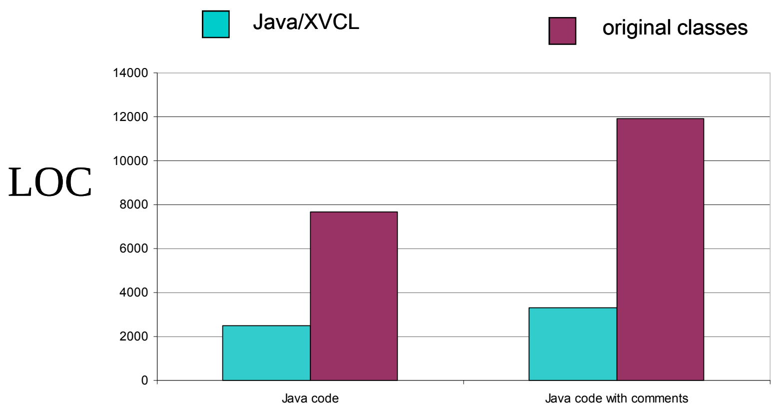

X-framework structure




SPC // specifies how to generate all the buffer classes
<set Type = Int, Short, Float, Long, Double, Char, Byte />
<set type = int, short, float, long double, char, byte />
<set elmntSize = 0, 1, 3, 2, 2, 3, 1 />
<adapt [T]Buffer.s />
<adapt Heap[T]Buffer />
…
<adapt ByteBufferAs[T]BufferR[B|L] />
x-frame [T]Buffer.s // specifies how to generate [T]Buffer classes
<while Type, type, elmntSize>
<select option = Type>
<option Char>
<adapt [T]Buffer />
<insert implements >
,CharSequence
<insert toString >
implementation of method toString() for CharBuffer
<insert extraMethods >
<adapt extra-methods-CharBuffer />
<option Byte>
<adapt [T]Buffer />
<insert extraMethods >
<adapt extra-methods-ByteBuffer />
<otherwise>
<adapt [T]Buffer />
</select>
</while>
<x-frame [T]Buffer outfile = @TypeBuffer.java >
public abstract class @TypeBuffer extends Buffer
extendsBuffer implements Comparable <break implements>
{ final @type[] hb;
Buffer(int mark, int pos, int lim, int cap,
@type[] hb, int offset) { … }
<break toString >
public String toString() {
StringBuffer sb = new StringBuffer();
sb.append(getClass().getName());
etc.
return sb.toString(); } }
<break extraMethods >
// methods specific to CharBuffer or ByteBuffer
}
x-frame extra-methods-CharBuffer
public static CharBuffer wrap(CharSequence csq) {
… }
x-frame extra-methods-ByteBuffer
public static ByteBuffer allocateDirect(int capacity)
{ return new DirectByteBuffer(capacity); }
CS6201 Set #3 SP and GD
© 2010 Stan Jarzabek
NUS
51


Propagation of customizations
<x-frame SPC >
<set Type = Int, Short, Float, Long, Double, Char, Byte />
<set type = int, short, float, long, double, char, byte />
<set elmntSize = 0, 1, 3, 2, 2, 3, 1 />
<while Type, elmntSize>
<select option = Type>
<option Char>
<adapt [T]Buffer>
<insert extraCodeForCharBuffer>
</while>
<adapt>
<x-frame [T]Buffer outfile = @TypeBuffer.java >
public abstract class @TypeBuffer extends Buffer
extendsBuffer implements Comparable <break implements>
{ final @type[] hb;
Buffer(int mark, int pos, int lim, int cap,
@type[] hb, int offset) { … }
<adapt slice>
}
<adapt>
<x-frame slice >
public @TypeBuffer slice() {
… (as before)
int off = (pos << @elmtSize);
<break extraCodeForCharBuffer />
…. }
CS6201 Set #3 SP and GD
© 2010 Stan Jarzabek
NUS
52

Java/XVCL vs. original classes



- LOC reduction: 68% Java code, 72% (code + comments)
- in Java/XVCL, we count both Java code and XVCL commands

How did we arrive at the XVCL solution?

1. Analyze the existing Buffer library
2. Identify groups of similar classes
3. Analyze similarities and differences in detail
4. Design meta-classes and parameters
5. Unify simple clones (similar methods and other similar fragments)
6. Refine the solution
7. Write the SPC
8. Generate classes and test the code

Maintenance effort: XVCL solution vs. the original code

- suppose we add Complex buffer

New classes	Changes in original Buffer library		Changes in XVCL solution	
	No.	type	No.	type
ComplexBuffer	25	automatic	3	manual
	17	manual		
	2	manual	2	manual
HeapComplexBuffer	21	automatic	3	manual
	10	manual		
HeapComplexBufferR	16	automatic	3	
	5	manual		

Technology assessment methods

1. benchmarking against an industry productivity index [QSM, Bassett]

conduct comparative studies:

2. controlled experiments
3. analytical argumentation
4. metrics-based evaluation of equivalent program solutions developed using different techniques

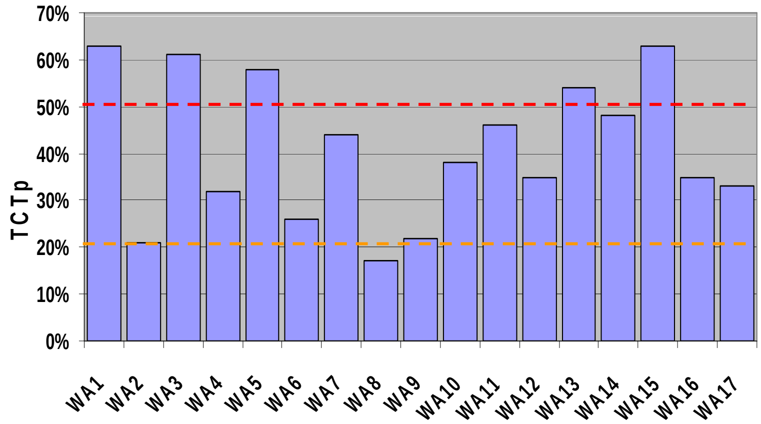
Clone phenomenon

High rates of cloning

percentage of repetitions	observed in:
20% – 50%	reengineering projects
68%	Java Buffer library
40% - 60%	STL in C++: containers and some functions
68% 50%	C# .NET; J2EE, command and control appl.
17% - 63%	17 Web Applications (simple clones)
60% - 90%	Web portals ASP (simple + structural) clones
80% - 95%	business applications in COBOL (reuse with frame technology)

Cloning in 17 Web Applications

% of code is contained in clones



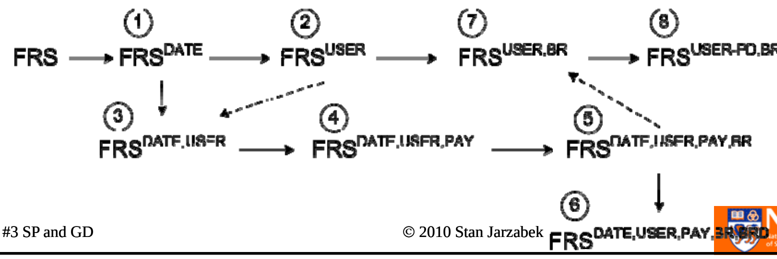
Situations that lead to cloning:

1. Poor design
2. Cloning to speed up development or maintenance
   – useful in short-term, may be harmful in long-term

   such clones often can be refactored
3. Avoiding clones compromises other important design goals
4. Cloning for performance or reliability reasons

Situations that lead to cloning:

5. Clones induced by the design technique
   - standardized architectures on JEE and .NET
   - pattern-driven development
6. Clones generated by IDEs
7. Software evolution
   - creating new system versions for new clients by copy and modify existing sources



Clones may be good or bad, depends how we look at it

On one hand:

- We need intentional clones, even tough they may contribute to maintenance cost
- Clones make similar parts of a program look similar
  - Standardization is good
- Some clones cannot be refactored

On the other hand:

- Clones can increase the risk of update anomalies
  - If change affects one clone instance, it may (or not) affect others

It is good know clone location and context

Why is it good to know clones?

- Software understanding and maintenance
  - improve comprehension of system structure
  - reduce update anomalies, control over ripple effects of changes, better traceability
- Re-engineering of legacy code into product lines for reuse (and also or easier maintenance)
  - recovery of recurring structures, candidates for reuse
- Helps in software quality assessment
- Similarity problem in non-software domains

Defining clones

- Two code structures of considerable size are clones of each other if there is significant similarity between them
- The actual size and similarity varies depending on the context, and is left to human judgment
- Clones may or may not represent program units that perform well-defined functions (syntactic clones)

Defining clones

- Are X and Y clones of each other?
- Similarity measures:
  - types of differences among X and Y
    - parametric differences
      - types, names, constants, operators, keywords, etc.
  - the minimum length of clone candidates
  - percentage of repeated code between X and Y
  - editing distance between X and Y
  - metrics-based measures

Similarity measures

- Editing distance measure
  - how easy is to edit fragment X to obtain Y?
  - the number of added/deleted lines between X and Y
  - the number of "atomic" editing operations on X to obtain Y from Y
- Metric-based similarity measure
  - compute metrics for X and Y
  - if they meet certain threshold - X and Y are considered clones of each other
  - e.g., # same tokens, complexity metrics, etc.

Defining structural clones

- Similarity measures for structural clones:
  - percentage of repeated code
  - similarity among elements of the two structures
  - similarity in relationships among elements of the two structures
  - difficult problem, depends on structural clone type
- "similarity" is subjective to expert judgment

Clone detection

Clone Miner: Basit, A.H. and Jarzabek, S. "Detecting Higher-level Similarity Patterns in Programs," *ESEC-FSE'05*, September 2005, Lisbon, pp. 156-165; CM/CA technology summary

Clone Miner (CM) detects similar program structures:

- first, CM detects simple clones
  - a token based clone detector, similar to CCFinder
- next, CM applies data mining techniques to find configurations of simple clones as *candidates* for structural clones

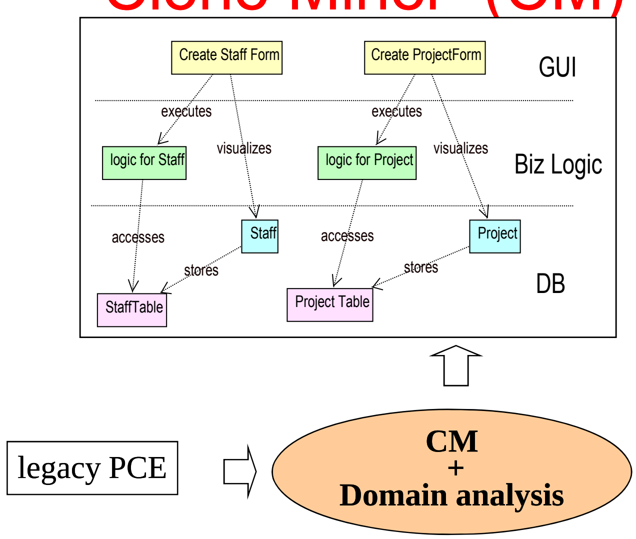
Clone Analyzer (CA)

- works with a user who decides which *candidates* represent meaningful design concepts

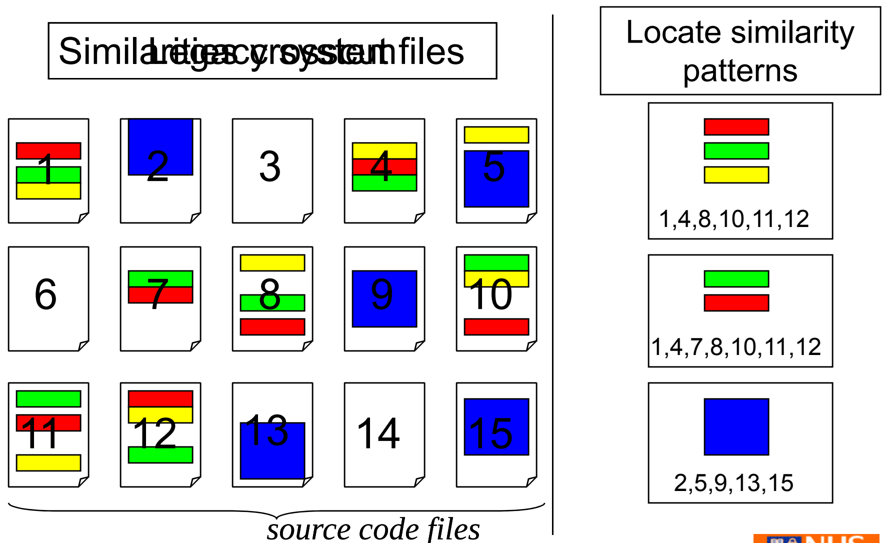
CM / CA tools

- Show the big picture of cloning situation
  - design level similarity patterns
- Work in interaction with an analyst
- Customizable
- Scalable
  - half a million lines of code ~ 3 minutes
- Extendible
  - C++, Java, …

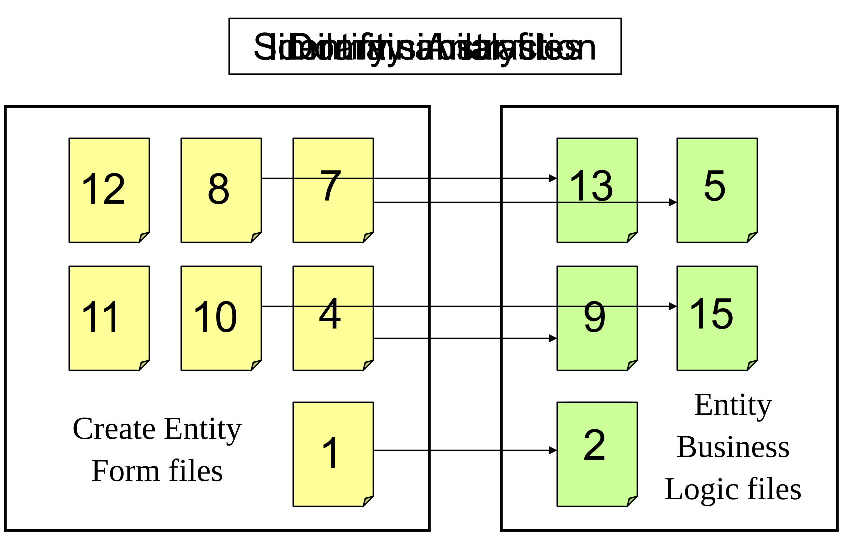
Clone Miner (CM)



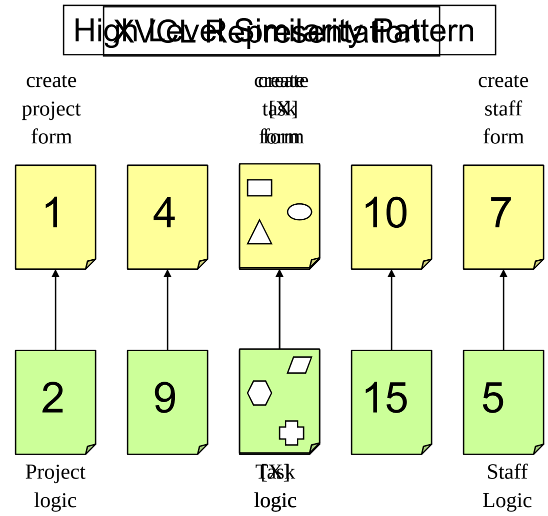
Clone Miner



Clone Miner



Clone Miner

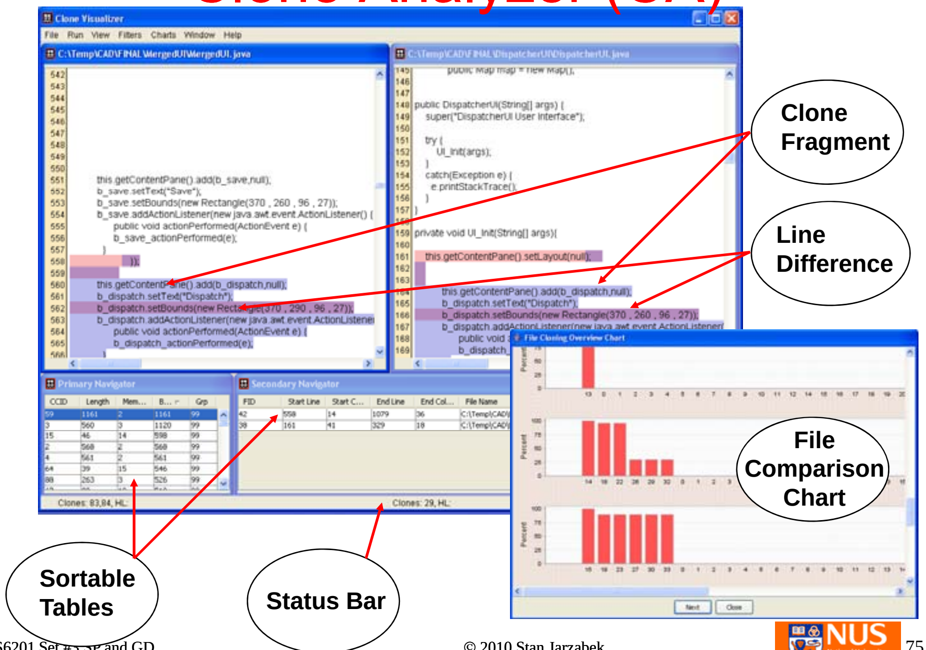


Clone Analyzer (CA)

CA provides interactive clone analysis features

- Sortable Tables integrated with Code Panels
  - basic access to clones detected by CM
  - comparison of clones (diff)
- Statistics View
- Clone-Query System
  - to filter focused views of cloning information
- Graphical presentations of cloning info

Clone Analyzer (CA)



Q & A

End of Set# 3 Similarity Patterns and Generic Design