Specification and Verification Challenges

Peter Müller

ETH Zürich

Switzerland

Joint work with Gary Leavens and Rustan Leino

Overview

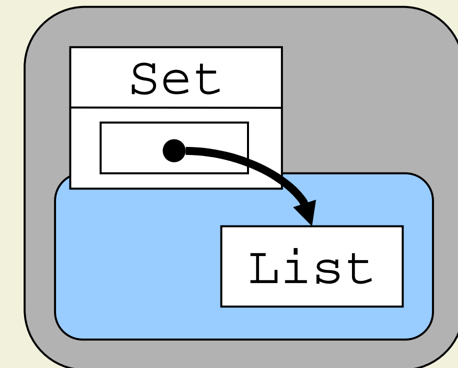
- Background
  - Sequential Java
  - JML-like specifications
  - Program verifiers such as Boogie, ESC/Java, Jive, KeY, etc.
- Emphasis: modular verification
  - Libraries
  - Scalability

Challenge Areas

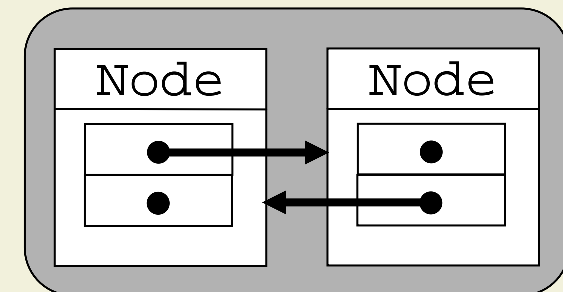
- **Heap structures**
- **Data abstraction**
- Frame properties
- Control flow
- Practical considerations

Ownership and Visibility Invariants

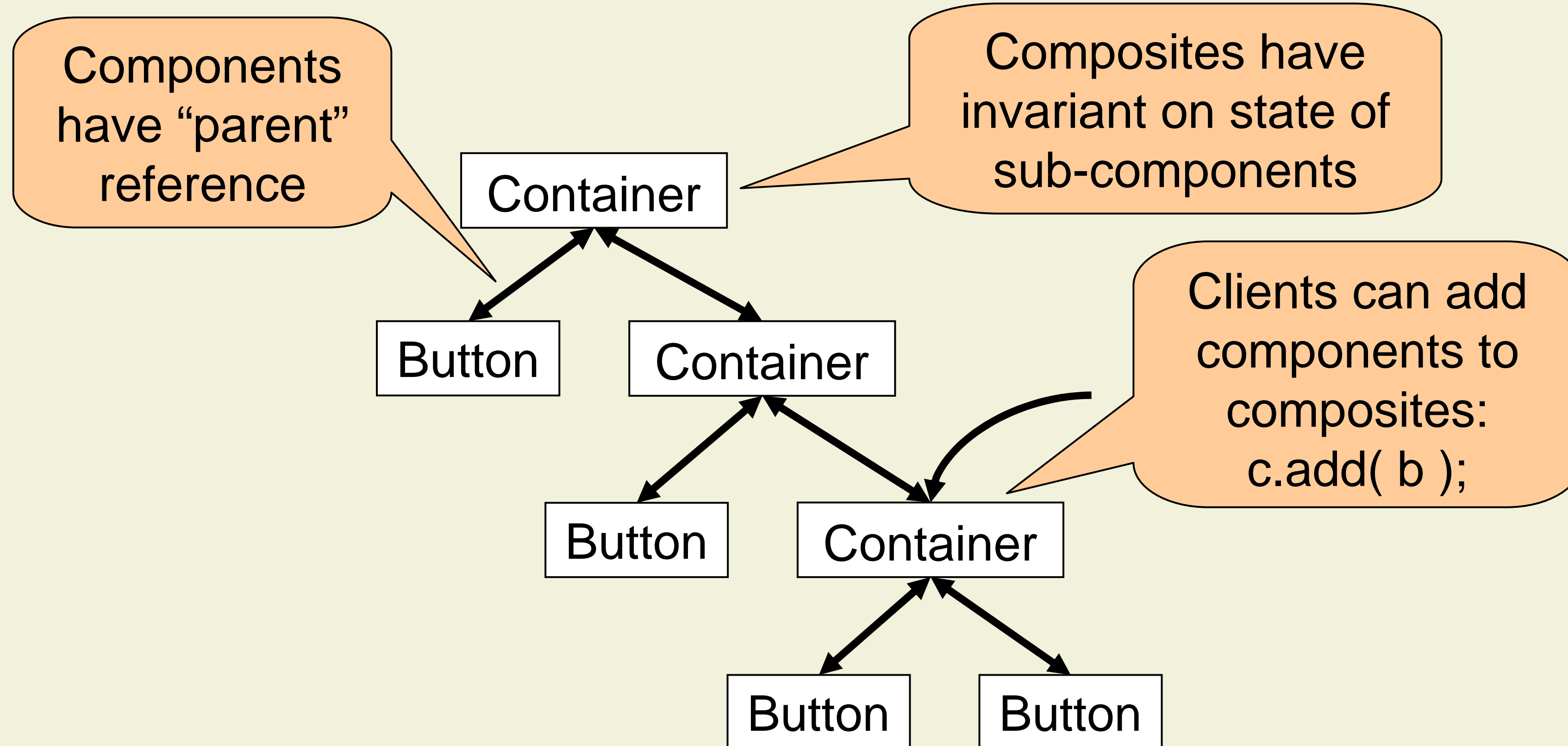
- Problem: for each state change, determine all invariants that are potentially affected
- Solution 1: ownership
  - Restrict invariants to owned objects
  - Prevent calls on “irrelevant” objects



- Solution 2: visibility
  - Restrict invariants to fields where invariant is visible
  - Impose proof obligations for visible invariants

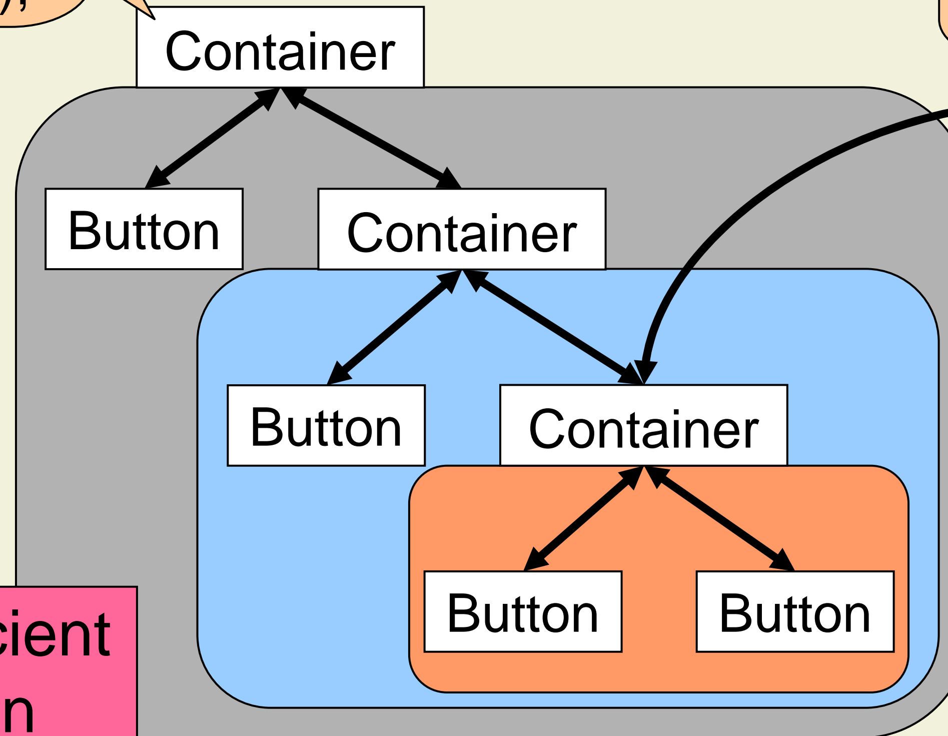


Composite Pattern: AWT Example



Composite Pattern: Ownership

Clients can add components to composites: root.add(b, c);



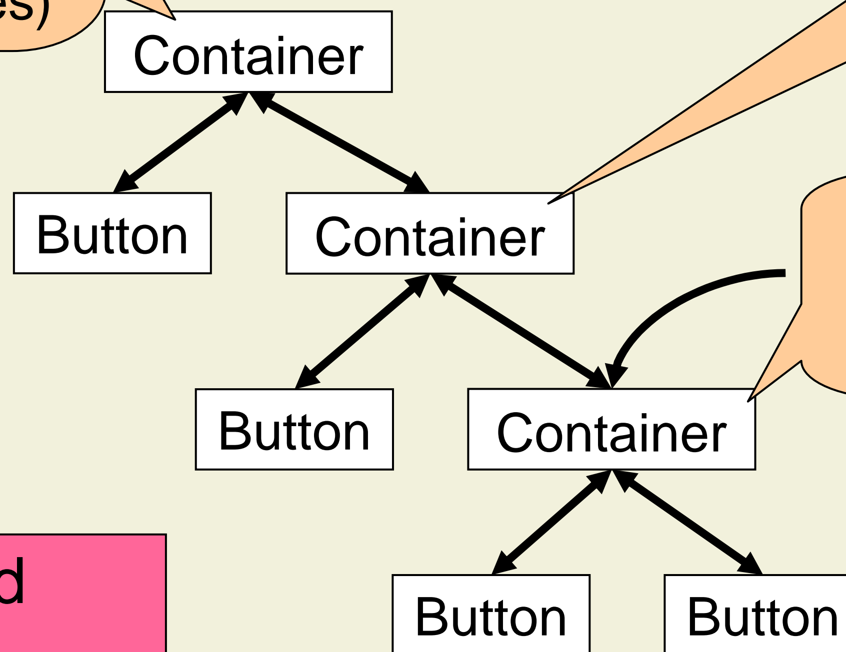
Direct modification is prevented

Unnatural, inefficient implementation

Composite Pattern: Visibility

Invariants must be visible in component class (no additional composite classes)

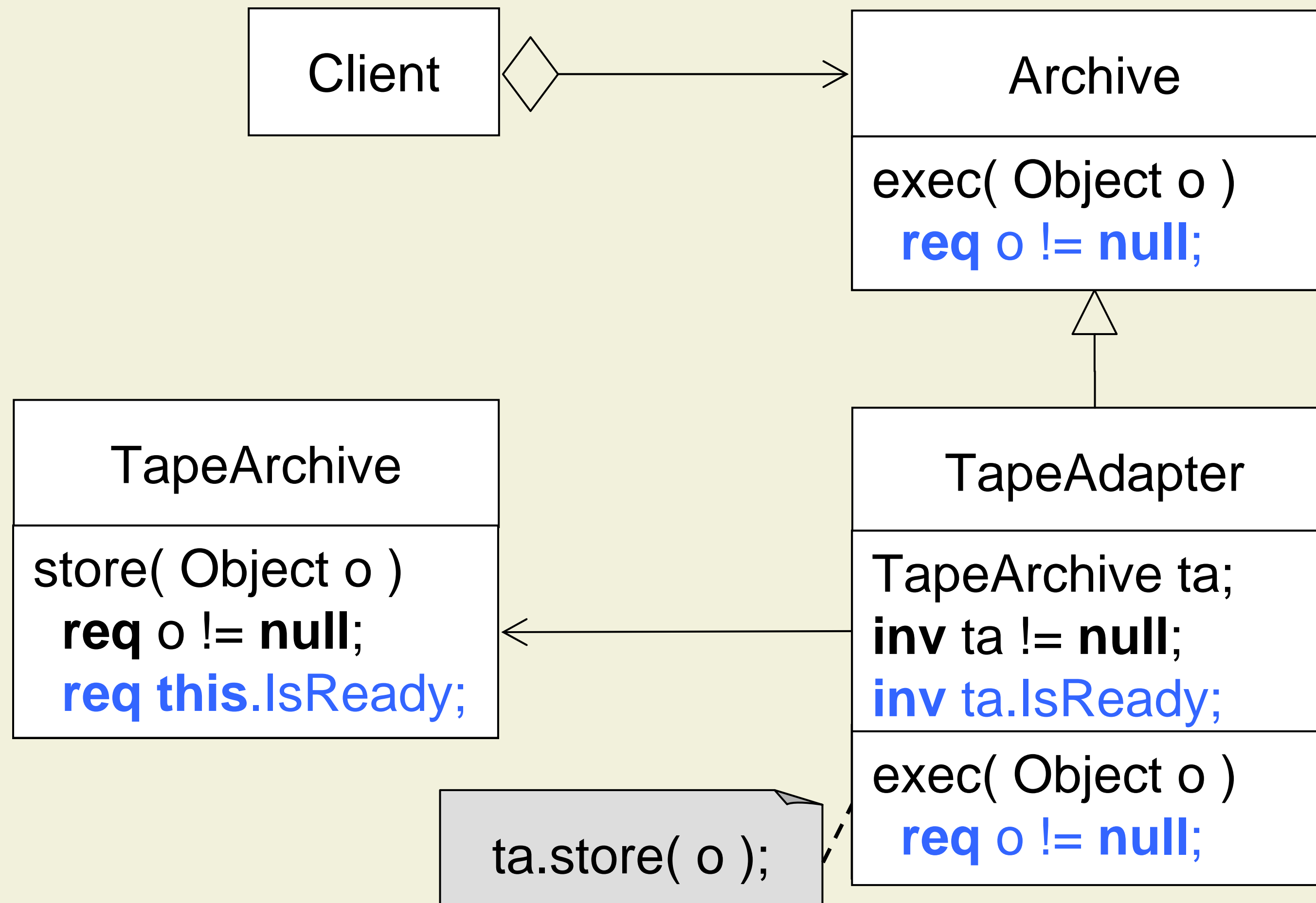
Visible state semantics prevents calls on parents



Proof obligations contain reachability predicates

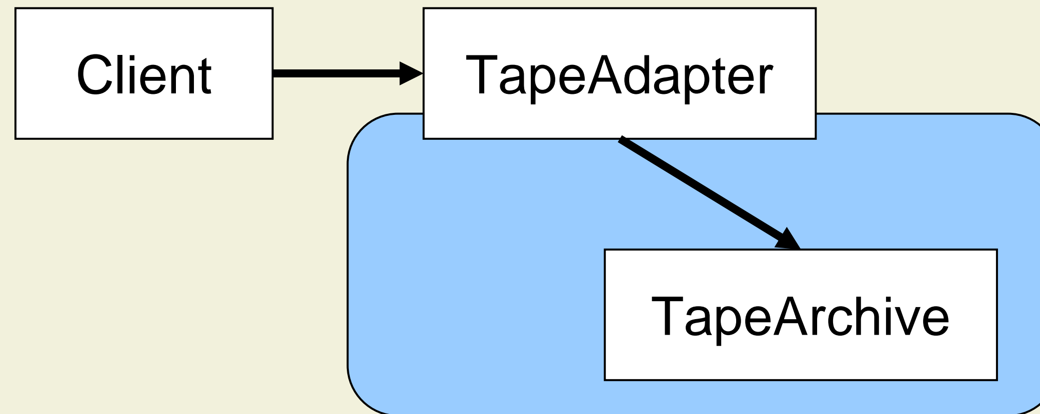
Complicated verification

Function Objects: Command Pattern



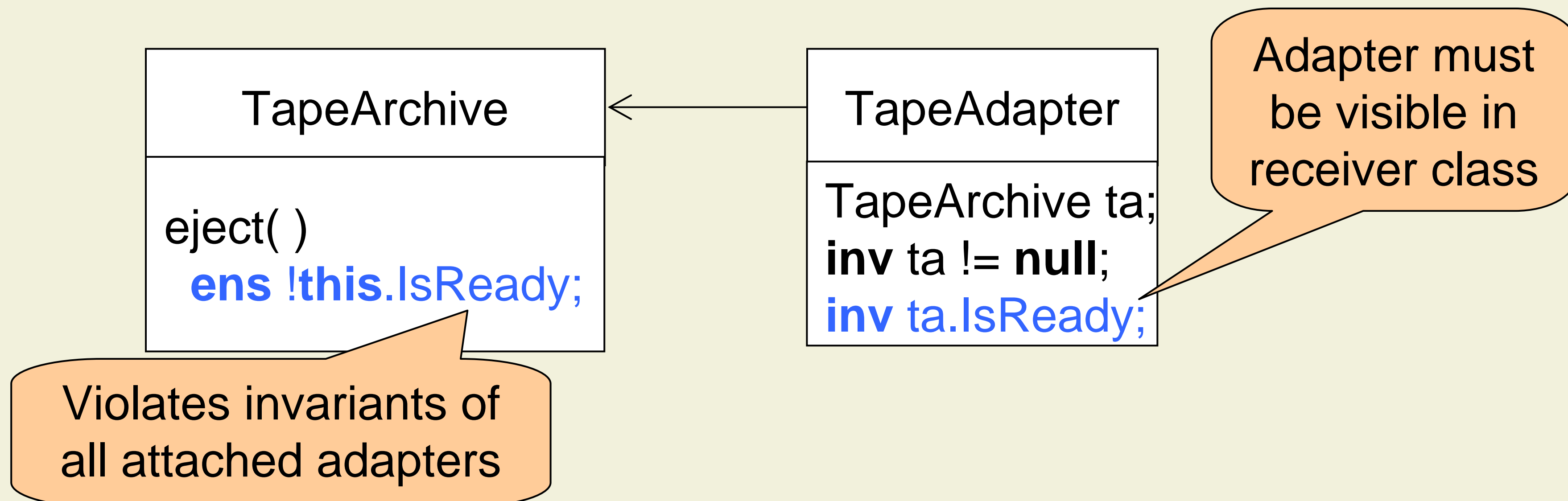
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Command Pattern: Ownership



- Receiver cannot be part of another ownership structure
- Only one function object per receiver
  - Impractical for, e.g., Observer Pattern

Command Pattern: Visibility



- No adapters for existing classes
  - But function objects are important to adapt legacy code
- Requires a way of disabling function objects

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Quantifiers and Comprehensions

```
static String[ ] filter( String[ ] args ) {
  int count = 0;
  for( int i = 0; i < args.length; i++ )
    if( args[ i ].startsWith( "-" ) ) count++;

  String[ ] out = new String[ count ];
  count = 0;

  for( int i = 0; i < args.length; i++ )
    if( args[ i ].startsWith( "-" ) ) { out[ count ] = args[ i ];  count++; }
  return out;
}
```

How to prove that count is within array bounds?

Loop Invariant with Quantifier

General quantifier not supported by automatic theorem provers

```
loop_invariant 0 <= count &&
  ( \num_of int j; 0 <= j && j < args.length; args[ j ].startsWith( "-" ) )
  == out.length - count;

for( int i = 0; i < args.length; i++ )
  if( args[ i ].startsWith( "-" ) ) { out[ count ] = args[ i ];  count++; }
```

Loop Invariant with Pure Method

```
requires 0 <= from;
ensures  a.length <= from ==> \result == 0;
ensures  from < a.length  ==>
   \result == ( a[ from ].startsWith( "-" ) ? 1 : 0 ) + countHits( a, from+1 );
pure static int countHits( String[ ] a, int from ) {
  int n = 0;
  for( int i = from; i < a.length; i++ )
    if( a[ i ].startsWith( "-" ) )  n++;
  return n; }
```

Non-trivial specification difficult for programmers. Automatic generation possible

```
loop_invariant 0 <= count  && countHits( args, i ) == out.length - count;
for( int i = 0; i < args.length; i++ )
  if( args[ i ].startsWith( "-" ) ) { out[ count ] = args[ i ];  count++; }
```

Lessons Learned from our FACJ Paper

- Program verifiers have come a long way
  - Many challenges have been (partially) solved, e.g., aliasing, callbacks, abrupt termination, etc.
- Current challenges occur in non-trivial, but common implementations (e.g., design patterns)
  - We did not consider concurrency, reflection, dynamic class loading
- Several (partial) solutions appeared while we were working on the paper
  - For instance, method calls in specs, model classes
- Good news: we are not running out of work!