Clone, Adapt and Improve!

Stratego User Days 2005

May 3.
Utrecht, The Netherlands

Anya Helene Bagge[1,2]
Martin Bravenboer[2]
Karl Trygve Kalleberg[1,2]
Koen Muilwijk[2]
Eelco Visser[2]

[1)]University of Bergen
Norway

[2)]Utrecht University
The Netherlands

Outline

- *Introduction*
- *Reuse and Composition*
- *The `clone` operator*
- *Adaptation with Aspects*
- *Case Studies*
- *Discussion*

Introduction

- *Reuse* is key to *efficient development* and *maintenance* of software systems.
- Existing reuse mechanisms, and in particular *inheritance*, falls disappointingly short of its promise.
- We sketch a slightly alternative approach to reuse, based on the idea of *software composition*, i.e., *compositing software* from *small, reusable parts*, while *adapting* them at composition time.

Reuse

- *Reuse* is dependent on techniques for *extension*, which in turn are dependent on techniques for *modularization*.
- Intuitively, reuse is about composing a system from reusable parts, but it seems difficult to decide on:
  - how the basic parts must be constructed; and
  - what the composition language should be like.
- *Anticipation of future need*, plays a crucial role in reuse. All too often, reuse is done in a boilerplate fashion, where code is copied verbatim, renamed and modified manually.

Extensibility and Modularization

- Aspects
- Mixins
- Traits
- Templates
- Inheritance
- Open Classes
- Meta Object Protocols
- Module systems
- Higher-order hierarchies
- Callbacks, delegates, closures aggregation
- Libraries

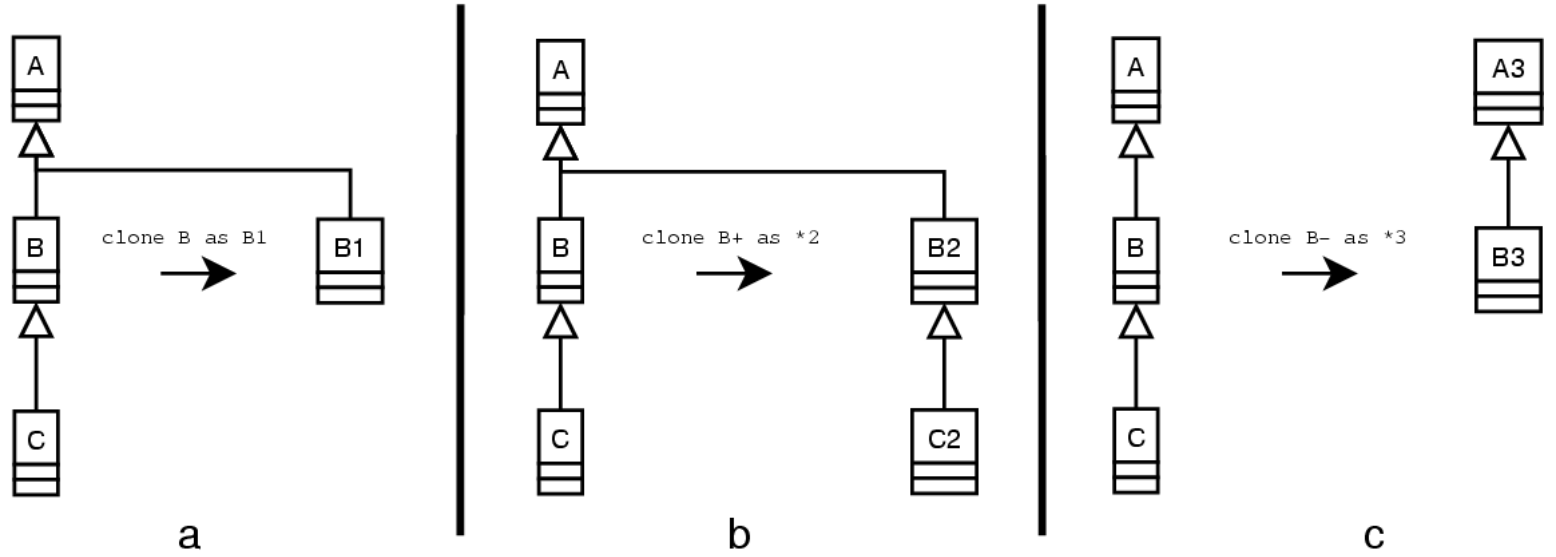
- Design Patterns
- Component Engineering
- Frameworks
- Invasive Software Composition

Invasive Software Composition

- In the Aßmann sense, *invasive software composition* is:
  - *a component-based way of constructing software from greybox components, where composition is by program transformation to parameterize, extend, connect, mediate and aspect-weave components.*
- Our technique falls into the same category, but is provided as a meta programming technique with a minimal linguistic footprint.
  - It accepts the premise that boilerplate reuse is good!

The `clone` operator

- Purpose: *to clone (and rename) a named definition.*
  - In Java: *packages*, *classes*, *fields* and *methods*.



The `clone` operator (2)

```
Clone      ::= Visibility? clone Definition
               Direction? as RenameExpr
               WithClause*
Direction  ::= + | -
WithClause ::= with Aspect
             | with Identifier = Identifier
             | with MethodSig as MethodSig
Definition ::= PackageName
             | ClassName
             | MethodName
RenameExpr ::= Identifier
             | Identifier *
             | * Identifier
             | Identifier * Identifier
Aspect     ::= aspect AspectBody
             | aspect AspectName
MethodSig  ::= Visibility Type MethodName(Type, ...)
Visibility ::= private | protected | package | public
```

The `clone` operator (3)

- Cloning hierarchies:

```
clone Foo+;
clone Foo-;
```

- Name rewriting

```
clone Foo as Bar;
clone Foo- as ClonedFoo*;
```

- Visibility

```
protected clone Foo+;
private clone Foo-;
```

Example: Class hierarchy cloning

```
private clone Component+ as Cloned*;
abstract class Component {
  public final int MODAL = 1;
  private int state = 0;

  public abstract int getWidth();
  public abstract int getHeight();
  protected void setState(int s)
  { state = s; }
  public boolean isModal()
  { return state&MODAL; }
  public void paint() { ... };
}
class Message extends Component {
  private String m;
  public Message(String m)
  { this.m = m; setState(Component.MODAL); }
  public Message() {}
  public void setMessage(String m) { ... }
  public int getHeight() { ... }
  public int getWidth()  { ... }
  public void paint() { ... }
}
```

⇒

```
class ClonedComponent { ... }

class ClonedMessage
  extends ClonedComponent {
  ...
  public Message(String m) {
    this.m = m;
    setState(ClonedComponent.MODAL);
  }
  ...
}
```

Adaptation: Intertype Declarations

- Purpose: *Offer benefits of open classes; the parents of a class may be declared later, separately from the class definition.*
- Intertype declarations follow some rules:
  - Can *add any interface* to any class, provided an implementation for the interface is also added.
  - Can redeclare the parent of class, provided the new is a subclass of the old.

Adaptation: Intertype Declarations (2)

```
class SpecialComponent extends Component { ... }
class Border extends Component { ... }

aspect WithSpecialComponent {
  declare parents: Border extends SpecialComponent;
}
```

```
class SpecialComponent extends Component { ... }
class Border extends SpecialComponent { ... }
```

Adaptation: Aspects

- Purpose: *To adapt a cloned definition using fine-grained code composition.*
- When aspects are applied to a definition, they are applied once and system-wide.
  - Prohibits generative programming.
  - Cloning alleviates this.

Adaptation: Aspects

```
clone Message as LoggedMessage with aspect {
   pointcut messageCreation(String m) :
    initialization(LoggedMessage.new(String))
    && args(m);

  before(String m) : messageCreation {
    Logger.log(m);
  }
}
```

"Case Study": Implicit Parameterization of Classes

```
class Array {
  private Element data[];

  public Element getElement(int index) { ... }
  public void setElement(Element elt, int index)
  { ... }
}
```

```
clone Array as StringArray with Element = String;
```

```
class StringArray {
  private String data[];

  public String getElement(int index) { ... }
  public void setElement(String elt, int index)
  { ... }
}
```

Case Study: Implicit Parameterization of Classes

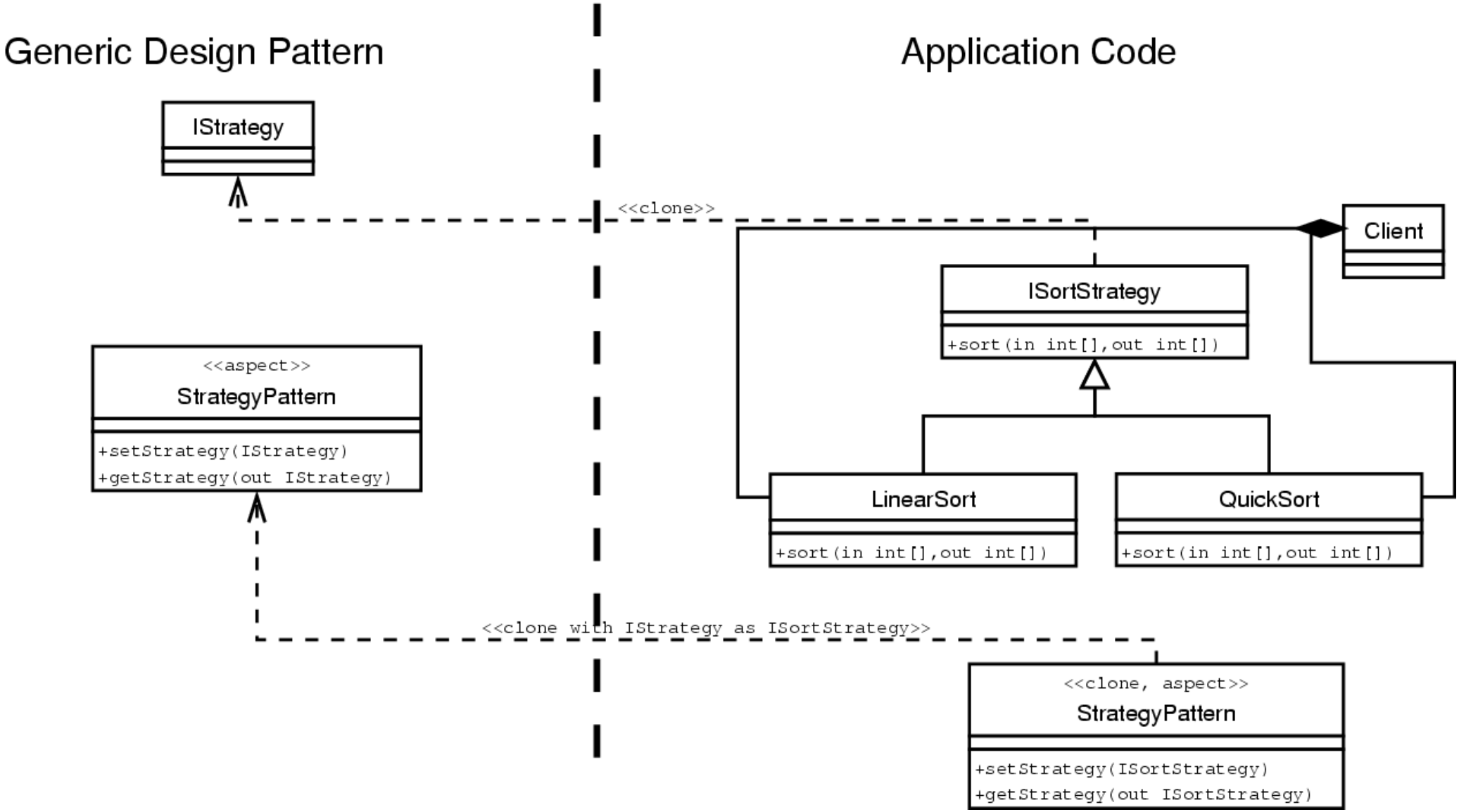
```
class Array {
  private Element data[];

  public Element getElement(int index) { ... }
  public void setElement(Element elt, int index)
  { ... }
}
```

```
clone Array as GenericArray<T> with Element = T;
```

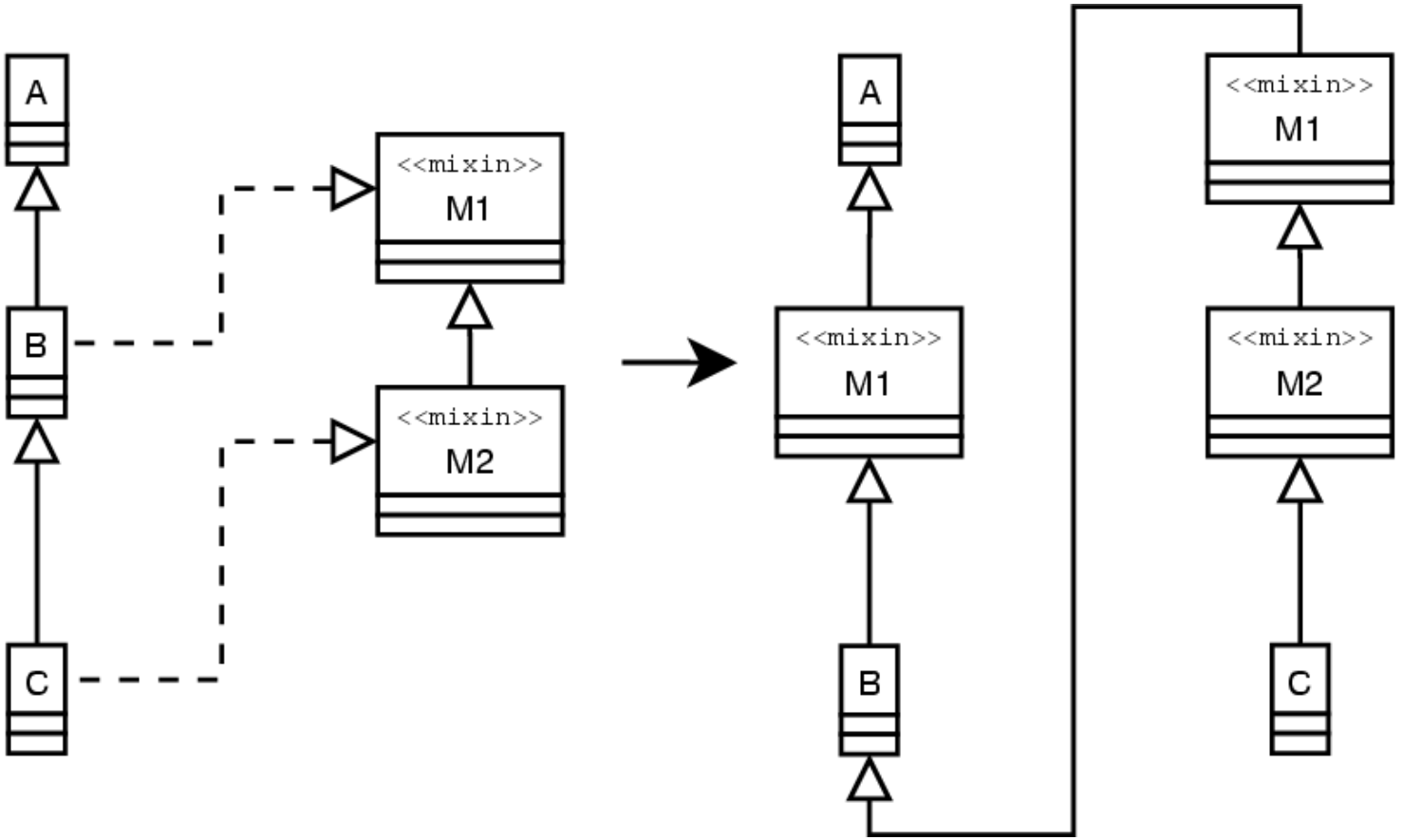
```
class GenericArray<T>{
  private T data[];
  public T getElement(int index) { ... }
  public void setElement(T elt, int index)
  { ... }
}
```

Case Study: Design Patterns



Case Study: Implementing Mixins

- *A mixin is an abstract subclass.*



Case Study: Implementing Mixins

- Mixins require three principal features of the language
  a) the ability to *clone* a *class* definition;
  b) the ability to *redeclare* the clone's *parents; and*
  c) a way to *linearize* the resulting *inheritance graph*
- We already have *(a)* and *(b), (c)* can be expressed as a small meta program in Stratego.

Case Study: Implementing Mixins

```
mixin Border {
  private void paintPreBorder() { ... }
  private void paintPostBorder() { ... }
  public void paint() { paintPreBorder();super.paint();paintPostBorder(); }
}

mixin ShadowedBorder extends Border {
  private void paintPreShadow() { ... }
  private void paintPostShadow() { ... }
  public void paint() { paintPreShadow();super.paint();paintPostShadow(); }
}

mixin NoisyComponent {
  private void emitSound() { ... }
  public void paint()
  { super.paint(); emitSound(); }
}

class FancyMessage extends Message mixes ShadowedBorder, NoisyComponent
{ FancyMessage(String m) { setMessage(m); } }
```

Case Study: Implementing Mixins

```
abstract class Border {
  public void paint()
  { ... ; super.paint(); ... }
}

abstract class ShadowedBorder extends Border { ... }
abstract class NoisyComponent { ... }

clone ShadowedBorder- as FancyMessage*;
clone NoisyComponent as FancyMessage*;

aspect MixBorderWithBorderedMessage {
  declare parents:
    FancyMessageBorder extends Component;
  declare parents:
    FancyMessageNoisyComponent extends FancyMessageBorder;
  declare parents:
    FancyMessageBorder extends FancyMessageNoisyComponent;
}
```

Discussion

- Cloning *decouples code reuse from inheritance*.
- In the presence of aspects, it *complements inheritance*.
  - (without aspects or other adaptation mechanism, it is less potent).
  - This gives us a form of *declarative boilerplate reuse*.
- The cloning language extension can be a *tiny DSEL* in a sufficiently powerful *meta programming language*.
  - Or expressed using a generic program transformation system.

Conclusion

- Is this *the* solution to *reuse*? No!
- As presented, it primarily makes sense in languages *with aspects*, but *without meta programming facilities*.