Abstract objects

With factoring, can introduce objects whose sole purpose is to be inherited from, not directly used or instantiated

- called **abstract** as opposed to **concrete** objects/classes
- may only be partially implemented;
  may call **abstract methods** that should be defined for concrete children

```
abstract object Point;
  -- [abstract method isn't legal Cecil]
  abstract method x(p@Point);
  abstract method y(p@Point);
  abstract method rho(p@Point);
  abstract method theta(p@Point);

  method area(p@Point) { p.x * p.y }
  method distance_to_origin(p@Point) { ... }
  method rotate(p@Point, amount) { ... }
  ...
```


Concrete implementations

```
template object CartesianPoint isa Point;
  field x(p@CartesianPoint) := 0;
  field y(p@CartesianPoint) := 0;
  method rho(p@CartesianPoint) { ... }
  method theta(p@CartesianPoint) { ... }
  method new_cartesian_point(x0, y0) {
    concrete object isa CartesianPoint {
      x := x0, y := y0 } }

template object PolarPoint isa Point;
  field rho(p@PolarPoint) := 0;
  field theta(p@PolarPoint) := 0;
  method x(p@PolarPoint) { ... }
  method y(p@PolarPoint) { ... }
  method new_polar_point(rho0, theta0) { ... }

concrete object Origin isa Point;
  method x(@Origin) { 0 }
  method y(@Origin) { 0 }
  method rho(p@Origin) { 0 }
  method theta(p@Origin) { 0 }
```


Object roles

Can declare whether an object is
`abstract`, `template`, or `concrete`

`abstract`:
- potentially incomplete
- can be inherited from
- cannot be used directly or instantiated

`template`:
- complete
- can be inherited from
- can be instantiated
- not to be used directly

`concrete`:
- complete
- can be inherited from
- can be instantiated
- can be used directly

Classless object model

Most OO languages distinguish objects/instances from classes, distinguish subclassing from instantiation

Cecil doesn't: **classless** (a.k.a. **prototype-based**) object model

Individual objects have their own implementation directly
- methods specialized directly on object, not separate class thing
- objects with methods/fields specializing on them act like classes

Individual object can inherit behavior from other objects
- if no additional customization, then act like an instance
- if new/overriding implementation, then act like a subclass

Class-like objects can be used directly as instances, too
- e.g. the `Origin` object
- useful for constants, enumerated values, `true`, `false`, etc.

Use object creation expression w/ optional initialization instead of special constructor operation

Class-based model

If have explicit classes:

- put field and method declarations with classes
  - class methods vs. instance methods,
    class fields vs. instance fields
- `new C`: creates instances of class *C*
- constructor methods in class called by `new`
- `this`/`self` to refer to implicit receiver of instance methods

```
class Point {
  var field x := 0;
  var field y := 0;
  method area { x * y }      -- self.x * self.y
  method shift(dx, dy) {
    x := x + dx;       -- self.x := self.x + dx;
    y := y + dy; }
  constructor(x0, y0) {
    x := x0; y := y0 }
  class field the_origin := new Point;
  class method origin() { the_origin }
}
let p := Point.origin();
p.shift(3,4);
```


Metaclasses

In pure, class-based OO languages (e.g. Smalltalk):

- every thing is a first-class object
- every object has a class

⇒ every class has a class

What's the class of a class? a **metaclass**

- stores the methods to handle messages sent to the class (e.g. `new`, other class methods)

What's the class of a metaclass?

Smalltalk-76 and Smalltalk-80 use different mechanisms on how to structure the metaclass instantiation hierarchy & break the infinite regress of metaclasses

- Smalltalk-80 more expressive (allows interesting class methods), but more complicated structure
- metaclasses are hardest part of ST to learn
  ⇒ led in part to development of classless languages

Multiple dispatching

If we can specialize first argument for particular classes, why not any subset of arguments?
⇒ **multi-methods**

```
method =(p1@Point, p2@Point) {
  p1.x = p2.x & { p1.y = p2.y } }

method =(p1@ColoredPoint, p2@ColoredPoint) {
  resend & { p1.color = p2.color } }
```

Resolve overloading by:

- find all applicable methods
- select & invoke single most-specific application method

```
let p1 = new_point(...);
let p2 = new_point(...);
let cp1 = new_colored_point(...);
let cp2 = new_colored_point(...);

print(p1 = p2);  -- only Point×Point applies
print(p1 = cp2); -- ditto
print(cp1 = p2); -- ditto
print(cp1 = cp2);-- both apply, CP×CP wins
```


Multi-method overriding

One mm overrides another if:

- for all the other's specializers,
  the first method's corresponding specializers are equal to or descend from the other's, and
- either:
  - at least one of the first's specializers strictly descends from the other's, or
  - one of the first method's formals is specialized while the other's is not

```
method foo(p1@Point, p2@ColoredPoint)
  overrides
method foo(p1@Point, p2@Point)

method foo(p1@ColoredPoint, p2@ColoredPoint)
  overrides
method foo(p1@ColoredPoint, p2)
```

Ambiguous methods

Two methods may be mutually ambiguous:
neither overrides the other

```
method foo(p1@Point, p2)
  ambiguous with
method foo(p1, p2@Point)

method foo(p1@ColoredPoint, p2@Point)
  ambiguous with
method foo(p1@Point, p2@ColoredPoint)
```

It's an error to send a message and find no single most specific application method

- no applicable methods: message not understood
- no single most specific method: message ambiguous

Resolving ambiguities

Can resolve ambiguities by defining an overriding method

- method can do its own thing, or
  it can use directed resend to invoke one or more of existing methods

```
method foo(p1@ColoredPoint, p2@Point) { ... }
method foo(p1@Point, p2@ColoredPoint) { ... }

method foo(p1@ColoredPoint, p2@ColoredPoint) {
  -- select first foo method above:
  resend(p1, p2@Point) }
```


Evaluation of multi-methods

Advantages:

Multi-methods unify & generalize:

- top-level procedures
  (no specialized arguments)
- regular singly dispatched methods
  (specialize first argument)
- overloaded methods
  (resolve overloading dynamically, not statically)

Multi-methods naturally allow existing objects/classes to be extended with new behavior

Problems:

What's the programming model?
What's the encapsulation model?

How to typecheck uses & definitions of multi-methods?
How to implement efficiently?

Multiple inheritance

Can inherit from several parent objects:

```
object Shape;
object Rectangle isa Shape;
object Rhombus isa Shape;
object Square isa Rectangle, Rhombus;
```

Subclass gets union of all fields and methods inherited from all parents

Ambiguities

Can get ambiguities due to MI, just like with MMs,
e.g. if two superclasses define methods,
neither of which overrides the other

```
object Rectangle isa Shape;
  method area(r@Rectangle) { ... }
object Rhombus isa Shape;
  method area(r@Rhombus) { ... }
object Square isa Rectangle, Rhombus;

let s := new_square(4);
... area(s) ...   → ambiguous!
```

Can resolve ambiguities by adding overriding method,
just like with MMs

```
  method area(s@Square) {
    resend(s@Rectangle) }
```


Diamond-shaped inheritance

How to determine method overriding if parent object is reachable along multiple inheritance paths?

```
object Shape;
  field center(s@Shape);
  method is_shape(s@Shape) { ... }
  method is_rectangular(s@Shape) { ... }
object Rectangle isa Shape;
  method is_rectangular(r@Rectangle) { ... }
object Rhombus isa Shape;
object Square isa Rectangle, Rhombus;

let s := new_square(4);
... is_shape(s) ...         → ambiguous?
... is_rectangular(s) ...   → ambiguous?
... center(s) ...           → ambiguous?
```

Different languages resolve these questions differently,
or don't support multiple inheritance at all

Mixins

MI may match natural classification schemes well

MI enables new programming idioms, e.g. **mixins**

- highly factored abstract objects
- often independent axes of mixins
- each concrete object combines one mixin choice from each axis as part of its behavior

Example axes for shapes in a user interface:

- colored or not, bordered or not, titled or not, ...
- method for handling mouse clicks
- ...

```
object CheckBox
  isa Square,
      ColoredShape,
      BorderedShape,
      ShapeWithIcon,
      ClickableShape,
      ...;
```

Often have diamond-shaped inheritance patterns

Encapsulation

How to hide internal implementation details?

Design for Cecil:

- wrap declarations in a `module`
- annotate declarations as `public` or `private`
- `import` other modules, see only public decls

```
module PointMod {
  object Point;
  public get private set var field x(@Point);
  public get private set var field y(@Point);
  public method new_point(x0,y0) { ... }
  private method ...
}
module ShapeMod {
  public import PointMod;
  private import ShapeHelperMod;
  public object Shape;
  public field center(s@Shape);
  ...
}
```

Object/class-based encapsulation

An ADT-oriented alternative: `object` declaration is module

```
object Point {
  public get private set var field x;
  public get private set var field y;
  public constructor(x0,y0) { ... }
  private method ...
}
object Shape {
  public field center;
  protected field color;
  ...
}
object ShapeHelper {
  friend Shape;
  ...
}
```

`protected`: visible to subclasses but not regular clients

- hard to simulate in module-based encapsulation

`friend`: list objects that have private access

Not clear what analogue of `import` is

Summary, so far

Key OO ideas, across all OO languages

- ADTs: objects/classes, methods, fields, encapsulation
- method overloading, dynamic dispatching
- inheritance, method overriding
- idioms like factoring, abstract objects

Choices in dynamically typed OO languages, seen in Cecil

- class-based vs. classless model
- fields accessed directly or via messages
- single dispatching vs. multiple dispatching
- single inheritance vs. multiple inheritance
- class-based vs. module-based encapsulation
- presence of first-class functions