Secretary Pattern: An Alternative for Segregation of Behaviors

RENATO CORDEIRO FERREIRA, Universidade de São Paulo
ÍGOR BONADIO, Universidade de São Paulo
ALAN MITCHELL DURHAM, Universidade de São Paulo

A class with multiple behaviors increases coupling if separated subsystems of a program use different subsets of these behaviors. When behaviors share few data but reuse non-trivial code, it is difficult to increase encapsulation by breaking the class in components (as proposed by the COMPONENT pattern) and still avoid duplication. In order to address this situation, this paper introduces the SECRETARY pattern, where auxiliary classes – called **Secretaries** – represent the behaviors and the main class – called **Boss** – is derived from the original entity. A secretary keeps only the data and exposes only the methods related to its behavior. A boss implements all algorithms and holds all data that do not change when executing any of its behaviors. The secretaries receive all client's requests and delegate to their boss. This design decreases coupling, as clients use only the behaviors they need; and keeps cohesion, as all relevant code is kept in one class. As the boss remains immutable, algorithms can be executed with different set of parameters in parallel. Results can be pre-computed and cached in the secretary, speeding up the response time for known arguments. This pattern has been originally created in the refactoring of ToPS framework and can be applied to increase the reuse code in games.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features— *Patterns*

Additional Key Words and Phrases: Secretary Pattern, design patterns, segregation of behaviors

ACM Reference Format:

Ferreira, R. C., Bonadio, Í. and Durham, A. M. 2016. Secretary Pattern: An Alternative for Segregation of Behaviors. jn V, N, Article 1 (November 2016), 5 pages.

1. INTENT

This design pattern aims to decrease the coupling related to classes that have complex interfaces, this is, classes whose methods can be grouped in different semi-independent subsets. Here, "semi-independent" means methods conceptually independent but whose segregation into smaller classes would not increase the reuse of code. The SECRETARY pattern allow developers to create auxiliary objects called **Secretaries**) that keep mutable data and provide access to the behaviors of the main class (called **Boss**), moving client's dependencies to the auxiliary while keeping the relevant implementation in a single class.

2. MOTIVATION

A First Person Shooter (FPS) is a game that simulates a fight among players Each player carries a set of weapons that are used to injure its opponents. A general weapon has three major behaviors:

—***Shooting***: The capacity of calculating which objects in the scenario will be damaged by the weapon;
—***Targeting***: The capacity of simulating the point that the weapon may hit;
—***Reloading***: The capacity of filling the ammunition of the weapon.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. A preliminary version of this paper was presented in a writers' workshop at the 17th Conference on Pattern Languages of Programs (PLoP). SugarLoafPLoP'16, November 16-18, Buenos Aires, Argentina. Copyright 2016 is held by the author(s).

In this kind of game, there are many types of weapons. Handguns (which shot projectiles) and machine guns (which bursts) are the most common. Flamethrowers, cross-and-bows and boomerangs are more exotic. Each category has a specific algorithm to calculate their collision area (ray tracing for the first group, elliptical and curved routes for the second one). These algorithms are necessary for the *shooting* and *targeting* behaviors.

The simplest way to implement this part of the game is by creating one class for each category of weapon. However, there is a problem with this approach: each class has a complex public interface with different methods for their behaviors. Nonetheless, not all methods are useful for the clients of the classes: the rendering subsystem, for example, needs to access the *reloading* data to display the number of projectiles in the status bar and the *targeting* methods to show the path of the projectiles to the player; the physics subsystem, in turn, only needs the *shooting* methods to hit the objects, reducing their health. As a consequence, the weapon classes couples several domains of the game - a great source of problems in the point of view of interface segregation [Martin 2000].

Considering the drawbacks of increasing the coupling, it is clear that this approach is not ideal to create a robust implementation. Nevertheless, splitting the classes in components (as proposed by the Component pattern [Nystrom 2014]) would not beneficial as two of the behaviors (*shooting* and *targeting*) reuse code (the collision algorithm) and all of them need to access the parameters specific of each category of weapon (as different kinds of ammunition).

3. PROBLEM

How can we decrease the coupling of a system that has entities with many behaviors, but whose methods share some data and code?

4. FORCES

—**Cohesion:** All code can be written in a single class without the increase of coupling.

—**Reusability:** Common code can be accessed within a single class, facilitating refactoring [Fowler and Beck 1999] and the creation of more concise methods.

—**Simulation:** The main object remains immutable and the execution of behaviors can be simulated with different sets of parameters.

—**Parallelism:** Through moving shared data to outside the main object, tasks requested to it can be executed in parallel.

5. SOLUTION

The Secretary pattern proposes a way to address the segregation of behaviors of complex entities. An entity is an abstract idea of a real world object or concept. For each behavior, the pattern determines the creation of a **Secretary** that represents it. Secretaries provide a narrowed view of the entity, with access only to the methods that implement the behavior they represent.

The class that represents the entity is promoted to a **Boss**, keeping all the code that implemented its behaviors. Each method now receives an extra parameter: a reference to the secretary that represents the behavior it implements. All calls are made indirectly through the secretaries, which have methods that match the boss' and delegate to it. This requires secretaries to keep a reference to the boss they serve.

In order to keep the state of the boss immutable, a secretary stores all data that can change during the execution of one of its methods. At the same time, it can also collect any external data that may be necessary. The boss keeps only the parameters that are immutable with respect to all its implemented behaviors, getting the rest from the secretaries whenever necessary.

In the example presented in section 2, we can model each category of weapon as a boss and `Shooter`, `Targeter` and `Reloader` as secretaries. Each subsystem of the game gets only the secretary (*Shooting*, *Targeting* and *Reloading* behaviors) from a given boss (weapons), loosing the coupling between the weapon and the subsystems.

6. STRUCTURE

The SECRETARY pattern has two categories of classes:

—**Boss**: Entity with multiple behaviors that will be segregated. It holds all code related to the behaviors in its methods, which do not modify any data kept inside the boss.
—**Secretary**: Auxiliary that provides access to the methods and stores mutable data related to a behavior. It delegates all computation to a given boss. It may collect data provided by clients.

These elements and their relations are illustrated by Figure 1 using an UML class diagram. The architecture presents an abstract class `Boss` and two concrete implementations `Boss1` and `Boss2`. This hierarchy contains two behaviors (`A` and `B`) whose methods are accessed through `SecretaryA` and `SecretaryB`. All requests received are forwarded by their concrete implementations to the appropriate bosses.

7. DYNAMICS

The dynamics of the SECRETARY pattern is illustrated by Figure 2. Following the example presented in the last section, a `Client` interacts with a `Boss` through a `SecretaryB`, which both transmits the requests made by the `Client` and stores a set of outer parameters provided by it.

8. IMPLEMENTATION

A class or a set of classes are good candidates for the application of the SECRETARY pattern if they have four major characteristics:

—**Multiple behaviors:** The candidate class has a complex interface, whose methods can be logically grouped in different semi-independent subsets.
—**Clustered data:** The candidate class has a partition of the methods and of the mutable attributes of the class such that that each subset of methods use only a subset of the attributes.
—**Specialization:** The candidate class has particular implementations of its methods, different from other classes with the same set of behaviors.
—**Code reuse:** The candidate class has common code among different behaviors.

Given a candidate, the pattern can be implemented as follows: each behavior is represented by a secretary that keeps one subset of the methods and the corresponding attributes. The methods in a secretary have the same signature of the original methods of the entity. The original methods, in turn, receive an extra parameter: a reference to the secretary. The entity class is then called a boss. Methods in the secretaries are implemented by delegating the reference and other parameters to the boss. Methods in the boss keep their original implementation but retrieve their data from the secretary.

As all methods in the secretaries have similar implementations and replicate the signature of their equivalents in the boss, it is possible to use metaprogramming to apply this pattern. In C++, templates can be used to generate methods in the secretary automatically while In dynamic languages the same result can be achieved with reflection. The following listings show a Ruby and C++ implementation of the example of section 2. Listing 1 presents the usage of the boss and secretary classes. Listing 2 and 3 defines the bosses while Listing 4 display the secretaries. Listing 5 finally presents the classes and macros that implement the metaprogramming techniques, which could be distributed as libraries to make the use of the pattern easier.

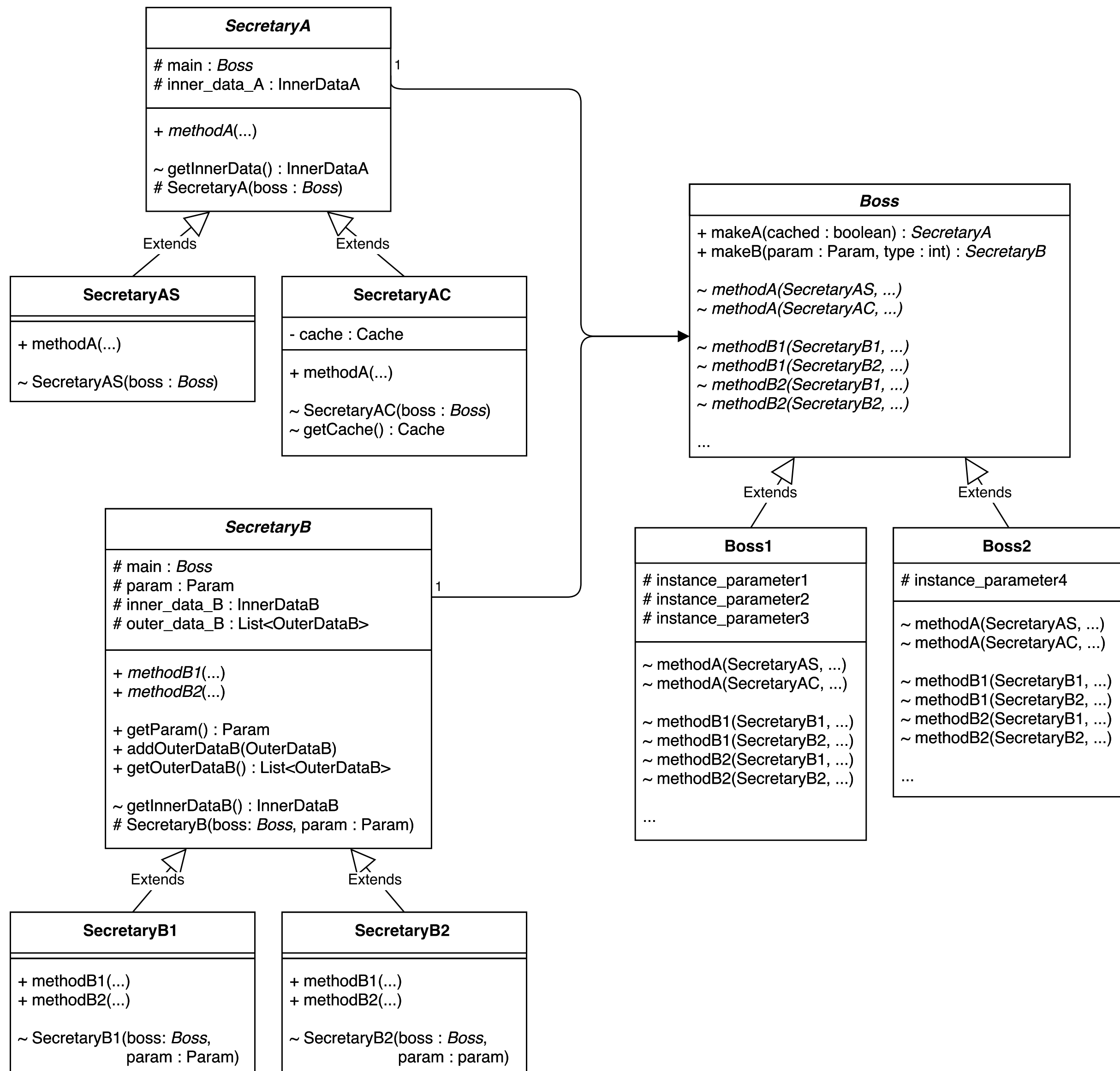

Fig. 1. **Class diagram of the Secretary pattern:** The `Boss` abstract class represents an entity with two behaviors: `A`, implemented by `methodA1` and `methodA2`; and `B`, implemented by `methodB1` and `methodB2`. These methods can be accessed through two secretaries: `SecretaryA`, which has a simple and a cached concrete implementation; and `SecretaryB`, which has two alternative concrete versions labeled 1 and 2. The concrete derived classes of `Boss` keep in their methods the specific code for each concrete secretary, receiving the corresponding class as first parameter. All inner data necessary for computations can be retrieved through getters. `SecretaryB`'s concrete children also have a common expansion of secretaries: they receive and store outer parameters (given by the client) that can be used by `Boss`'s children. Regarding the instantiation, `Boss` provides factory methods to construct new secretary objects by binding them with their creator object. Methods of behaviors in the `Boss` hierarchy and getters for the inner data are kept with package access. Methods of behaviors in the secretaries, factory methods in `Boss` and accessors to the outer data are made public for clients.

9. CONSEQUENCES

Beyond the forces already discussed, this pattern makes refactoring [Fowler and Beck 1999] easier, as all code of the behaviors can be potentially reused. As secretaries are very simple, programming languages that support inlining or method call optimizations can reduce the overhead of calling methods through secretaries to zero. When adding a method to a behavior, not all clients need to be recompiled: only the ones that use the corresponding secretary. This pattern can also be used to create secretaries for static methods. Secretaries can have their requests executed in parallel given that bosses remain immutable. They also provide a good place to cache the results generated by the bosses (as shown in Figure 1). The use of this pattern generates a design that follows the Interface Segregation Principle [Martin 2000], making clients do not depend on methods they do not use.

10. RELATED PATTERNS

This pattern was initially derived from the BRIDGE pattern [Gamma et al. 1995], which aims to decouple interfaces from implementations. In terms of applicability, it can be used in some situations that are not optimally addressed by the COMPONENT pattern [Nystrom 2014]. Lastly, the `Secretary` pattern can also be applied with other design patterns such as VISITOR and BUILDER [Gamma et al. 1995]. Examples of this can be found in the implementations presented in the next section.

11. KNOWN USES

The SECRETARY pattern was originally developed in the refactoring of **ToPS** (*Toolkit for Probabilistic Models of Sequences*), an object-oriented framework written in C++ which facilitates the integration of probabilistic models for sequences over an user defined alphabet [Kashiwabara et al. 2013]. The code can be found at: `https://github.com/topsframework/tops`. A simple C++ implementation containing generic entities and using template metaprogramming can be found at: `https://github.com/topsframework/tops-architecture`.

REFERENCES

FOWLER, M. AND BECK, K. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley. 2, 5

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design patterns: elements of reusable object-oriented software.* Vol. 47. Addison-Wesley Longman Publishing Co., Inc. 5

KASHIWABARA, A. Y., BONADIO, I., ONUCHIC, V., AMADO, F., MATHIAS, R., AND DURHAM, A. M. 2013. ToPS: A Framework to Manipulate Probabilistic Models of Sequence Data. *PLoS Computational Biology 9,* 10, e1003234. 5

MARTIN, R. 2000. Design principles and design patterns. *Object Mentor*. 2, 5

MEYERS, S. 2014. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++ 11 and C++ 14* 1st Ed. O'Reilly Media. 11

NYSTROM, R. 2014. Game programming patterns. 2, 5

STROUSTRUP, B. 2013. *The C++ Programming Language, 4th Edition.* Addison-Wesley. 11

Received February 2009; revised July 2009; accepted October 2009

SugarLoafPLoP'16, November 16-18, Buenos Aires, Argentina. Copyright 2016 is held by the author(s).

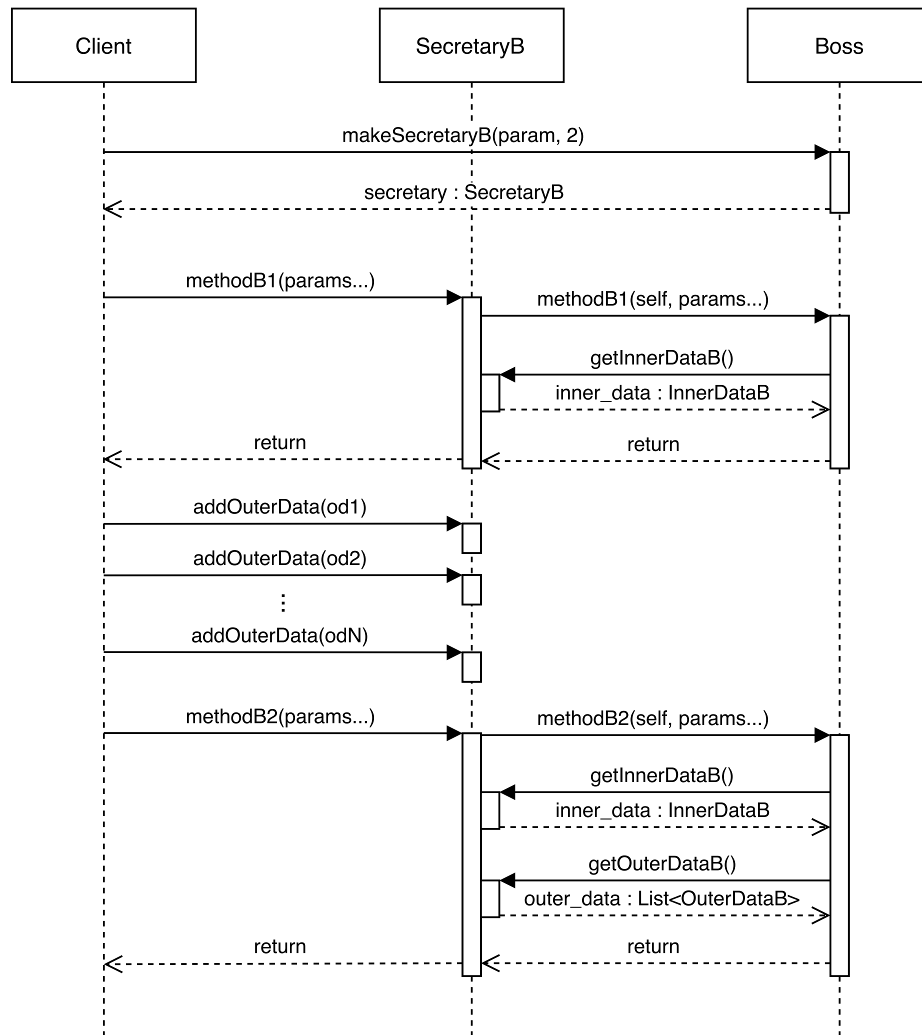

Fig. 2. **Sequence diagram of the Secretary pattern:** The `Client` object requests to the `Boss` object the access to a `SecretaryB` object. It then uses this secretary to call both `methodB1` and `methodB2`. In both cases, the secretary forward the request to the boss, passing as first parameter a reference to itself. The boss uses this reference to access the inner data stored inside the secretary. In `methodB2`, it also uses some outer data provided by the client to the secretary before the call.

```ruby
render = RenderComponent.new
sound = SoundComponent.new

hg = Hangun.new

render.render_path(
  hg.targeter, direction)

hg.shooter(:silver_bullet).shoot(
  direction)

reloader = hg.reloader(12)
reloader.reload(4)
sound.reload(reloader)

gd = Grenade.new

render.render_path(
  gd.targeter, direction)

gd.shooter(:flash_bang).shoot(
  direction)
```

```cpp
int main() {
  RenderComponent render;
  SoundComponent sound;

  Handgun mg;

  render.render_path(mg.targeter(), direction);

  mg.shooter(Projectile::silver)
     ->shoot(direction);

  mg.reloader(12)->reload(4);
  sound.reload(mg_reloader);

  Grenade gd;

  render.render_path(gd.targeter(), direction);

  gd.shooter(Projectile::incendiary)
     ->shoot(direction);

  return 0;
}
```

Lst. 1. **Usage of the boss and secretary classes, in Ruby and C++**. This example shows a simple FPS game with two types of weapons: handguns and grenades. The game has two components, one for rendering and other for sound playing. Weapons are implemented as bosses with two types of secretaries: shooters, responsible for finding the objects in the world scenario that will be damaged by the weapon; and targeters, responsible to provide the expected path of the projectiles. Handguns also have reloaders, whose aim is to store the number of projectiles available in a handgun.

```ruby
class Handgun < Boss
  secretary :shooter, [:shoot]
  secretary :targeter, [:path, :target]
  secretary :reloader, [:reload, :full?]

  def shoot(shooter, direction)
    trace = raytrace(direction)
    World.objects.each do |obj|
      if trace.colide?(obj)
        obj.hit(shooter.projectile)
      end
    end
  end

  def path(targeter, direction)
    raytrace(direction).points
  end

  def target(targeter, direction)
    raytrace(direction).last_point
  end

  def reload(reloader, ammunition)
    reloader.ammunition += ammunition
    if (reloader.ammunition >
        reloader.capacity)
      reloader.ammunition =
        reloader.capacity
    end
    reloader.ammunition
  end

  def full?(reloader)
    reloader.ammunition == reloader.capacity
  end

  def raytrace(direction)
    # computes ray tracing algorithm
  end
end
```

```cpp
class Handgun : public WeaponBoss<Handgun> {
 public:
  void shoot(Shooter<Handgun>* shooter,
             Direction d) {
    auto trace = rayTrace(d);
    for (auto object : World::objects())
      if (trace.collide(object))
        object->hit(shooter->projectile());
  }

  Point target(Targeter<Handgun>* targeter,
               Direction d) {
    return rayTrace(d).lastPoint();
  }

  vector<Point>
  path(Targeter<Handgun>* targeter,
       Direction d) {
    return rayTrace(d).points();
  }

  unsigned int
  reload(Reloader<Handgun>* reloader,
         unsigned int ammunition) {
    reloader->ammunition += ammunition;
    if (reloader->ammunition > reloader->capacity)
      reloader->ammunition(reloader->capacity);
    return reloader->ammunition;
  }

  bool isFull(Reloader<Handgun>* reloader) {
    return reloader->ammunition
           == reloader->capacity;
  }

 private:
  RayTracer rayTrace(Direction d) {
    // computes ray tracing algorithm
  }
};
```

Lst. 2. **Handgun Boss, in Ruby and C++**. This boss implements behaviors defined by three secretaries: `Shooter` (shoot), `Targeter` (calculate path of projectiles and target point) and `Reloader` (update the number of projectiles and check if the weapon is full). In both languages, the classes inherit from an auxiliary class. In Ruby this class is generic and works for any type of boss. It provides the method `secretary` that allows the creation of a secretary class from a symbol with its name and a list of symbols with the name of the methods that will be accessed through it. It also creates the factory method to create the secretary through its boss. In C++ the auxiliary class is a specific implementation that works only for this example. It uses macros to implement factories methods for the three secretaries.

```ruby
class Grenade < Boss
  secretary :shooter, [:shoot]
  secretary :targeter, [:path, :target]

  def shoot(shooter, direction)
    World.objects.each do |obj|
      if explosion_area(parable_trace(
        direction).last_point).include?(
          obj.position)
        obj.hit(shooter.projectile)
      end
    end
  end

  def path(targeter, direction)
    parable_trace(direction).points
  end

  def target(targeter, direction)
    parable_trace(direction).last_point
  end

  def parable_trace(direction)
    # computes the parable trace algorithm
  end

  def explosion_area(point)
    # calculates the explosion area
  end
end
```

```cpp
class Grenade : public WeaponBoss<Grenade> {
 public:
  void shoot(Shooter<Grenade>* shooter,
             Direction d) {
    for (auto object : World::objects())
      if (explosionArea(
            parableTrace(d).lastPoint())
              .find(object->position()))
        object->hit(shooter->projectile());
  }

  Point target(Targeter<Grenade>* targeter,
               Direction d) {
    return parableTrace(d).lastPoint();
  }

  vector<Point> path(Targeter<Grenade>* targeter,
                     Direction d) {
    return parableTrace(d).points();
  }

 private:
  ParableTracer parableTrace(Direction d) {
    // computes parable tracing algorithm
  }

  Area explosionArea(Point p) {
    // calculates the explosion area
  }
};
```

Lst. 3. **Grenade Boss, in Ruby and C++**. This boss has a definition similar to `Handgun`, but it only implements the behaviors provided by the `Shooter` and `Targeter`. It is important to notice that in the C++ implementation `Grenade` inherits from `WeaponBoss`, which also provides a factory method for the `Reloader` secretary. This is not a problem since the factory methods are templates, which are only generated by the compiler when they are used. If the `Targeter` factory method is called and one of the methods of the created objects is used, the compiler will emit an error.

```ruby
class Shooter < Secretary
  attr_reader :projectile

  def initialize(boss, projectile)
    @projectile = projectile
    super(boss)
  end
end

class Reloader < Secretary
  attr_accessor :ammunition
  attr_reader :capacity

  def initialize(boss, capacity)
    @capacity = capacity
    @ammunition = 0
    super(boss)
  end
end
```

```cpp
template<typename Weapon>
class Shooter {
 public:
  // constructors

  DELEGATE(weapon_, shoot)

  Weapon* weapon_;
  Projectile projectile_;
};

template<typename Weapon>
class Reloader {
 public:
  // constructors

  DELEGATE(weapon_, reload)
  DELEGATE(weapon_, isFull)

  Weapon* weapon_;
  unsigned int capacity_ = 0;
  unsigned int ammunition_ = 0;
};

DEFINE_SECRETARY(Targeter, target, path);
```

Lst. 4. **Secretaries definitions, in Ruby and C++**. Simple secretaries can be automatically generated by the metaprogramming techniques. This is the case of `Targeter` in both implementations. In order to add data that will be stored in the secretaries, it is necessary to explicitly define the classes. In Ruby, the auxiliary parent class `Secretary` provides the methods that delegate the calls to its child's boss. In C++, it is not possible to loop through a list of methods. The macro `DELEGATE` is called once for every delegated method and hides the detail of its implementation.

```ruby
class Secretary
  def initialize(boss)
    @boss = boss
  end

  def self.action(name)
    define_method name do |*args|
      @boss.send(name, *([self] + args))
    end
  end
end

class Boss
  def self.secretary(secretary, methods)
    define_method secretary do |*args|
      klass = nil
      begin
        klass = Object.const_get(
          secretary.to_s.capitalize)
      rescue
        klass = Class.new(Secretary)
        Object.const_set(
          secretary.to_s.capitalize,
          klass)
      end
      methods.each { |m|
        klass.action(m) }
      klass.new(*([self] + args))
    end
  end
end
```

```cpp
#define FACTORY(Secretary, Boss, name)      \
  template<typename... Args>                \
  Secretary<Boss>* name(Args&&... args) {   \
    return new Secretary<Boss>(             \
      static_cast<Boss*>(this),             \
      std::forward<Args>(args)...);         \
  }

#define DELEGATE(delegate, function)        \
  template<typename... Args>                \
  decltype(auto) function(Args&&... args) { \
    return delegate->function(this,         \
      std::forward<Args>(args)...);         \
  }

#define DECLARE_SECRETARY(Secretary)        \
  template<typename Weapon>                 \
  class Secretary

#define DEFINE_SECRETARY(Secretary, m1, m2) \
  template<typename Weapon>                 \
  class Secretary {                         \
   public:                                  \
    // constructors                         \
                                            \
    DELEGATE(weapon_, m1)                   \
    DELEGATE(weapon_, m2)                   \
                                            \
    Weapon* weapon_;                        \
  }                                         \

template<typename Weapon>
class WeaponBoss {
 public:
  FACTORY(Shooter, Weapon, shooter)
  FACTORY(Targeter, Weapon, targeter)
  FACTORY(Reloader, Weapon, reloader)
};
```

Lst. 5. **Metaprogramming, in Ruby and C++**. The details of the metaprogramming vary a lot accordingly to the programming language. In Ruby, the static method `secretary` in `Boss` creates factory methods for the secretaries registered as symbols in its subclasses. It also calls the static method `action` provided by `Secretary`, which creates in the `Secretary`'s subclasses the methods that delegate to their bosses. In C++, the metaprogramming is made by a combination of macros and templates. `FACTORY` defines the factory method for a secretary called `Secretary` (one of the macro arguments) that delegates to `Boss` (another parameter), naming the method as `name` (the last parameter). This method has a variadic template [Stroustrup 2013] which allows it to perfect forward all its received arguments to the constructor of the boss [Meyers 2014]. `DELEGATE` uses similar techniques, but creates methods in the secretaries that are responsible to delegate to their bosses. `DECLARE_SECRETARY` and `DEFINE_SECRETARY` declare and define a simple secretary class with at most two methods. This can be generalized, but it is out of the scope of this work. Finally, `WeaponBoss` is the base class of all bosses used in this example, which was explained in Listing 3.