Secretary Pattern: Decreasing Coupling while Keeping Reusability

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

Clients that depend on a class with multiple behaviors are unnecessarily tied to the behaviors they do not use. One possible solution is to split the class in many, but this is not a good strategy if different behaviors share common code. This paper introduces the Secretary pattern, which decreases the client's coupling while keeping maximum reusability. In this pattern, auxiliary classes – called **Secretaries** – represent behaviors and interact with clients while the main class – called **Boss** – holds the reusable implementation code. The Secretary pattern has been originally identified in the refactoring of the ToPS framework and is also used to implement trait objects and interface objects in the Rust and Go compilers.

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

Additional Key Words and Phrases: Design patterns, coupling, reusability

ACM Reference Format:

Ferreira, R. C., Bonadio, Í. and Durham, A. M. 2016. Secretary Pattern: Decreasing Coupling while Keeping Reusability. HILLSIDE Proc. of Latin American Conf. on Pattern Lang. of Prog. 11 (November 2016), 11 pages.

1. INTENT

This design pattern aims to decrease the coupling between clients and classes that have complex interfaces and whose methods can be grouped in different semi-independent subsets. Here, "semi-independent" means methods conceptually independent but whose implementations share code. The Secretary pattern allow developers to create auxiliary objects (called **Secretaries**) that represent a single behavior of the main class (called **Boss**). This way, clients can move their dependencies to the smaller auxiliary classes while they keep reusing implementation within a single main 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. An ordinary weapon has three major behaviors:

—***Shooting***: Calculating which objects in the scenario will be damaged by the weapon;
—***Targeting***: Finding the point that the weapon may hit;
—***Reloading***: 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 11th Latin American Conference on Pattern Languages of Programs (SugarLoafPLoP). SugarLoafPLoP'16, November 16–18, Buenos Aires, Argentina. Copyright 2016 is held by the author(s). HILLSIDE 978-1-941652-05-3

In this kind of game, there are many kinds of weapons: handguns (which shot projectiles), machine guns (which bursts), flamethrowers, grenades, boomerangs and others. Each category has a specific algorithm to calculate its collision area (ray tracing, elliptical routes, curved routes. etc.). These algorithms are reused by the *shooting* and *targeting* behaviors.

The simplest way to implement the situation described above is as follows: for each category of weapon available, create one class with all code and data related to it. Though simple, this approach has a problem: clients have needless dependencies with the complex interface of the weapons, given that some clients use only a subset of the behaviors available. The rendering subsystem, for example, only needs to know the number of projectiles (associated with the *reloading* behavior) and their path in the scenario (calculated by the *targeting* behavior) to display them to the player. The gameplay subsystem, in turn, only needs to identify which enemies were hit in a shot to decrease their health (an action done through the *shooting* behavior). This way, both subsystems are unnecessarily coupled with behaviors that they do not use.

There are some alternatives to loosen the coupling generated by the approach presented above. This include separating each behavior in its own component (as proposed by the Component pattern [Nystrom 2014]) or binding different roles to a core accordingly to the needs of different clients (as proposed by the Role Object pattern). However, these techniques have no obvious place to put the algorithms shared by the implementations of the behaviors. As a consequence, it is more difficult to reuse or refactor [Fowler and Beck 1999] code in the system.

3. PROBLEM

How can we decrease the coupling between a client and a class with many behaviors, but still allow these behaviors to be implemented with maximum reuse of code?

4. FORCES

—**Coupling:** Clients can use auxiliary classes in order to depend only of the behaviors of the main class that they need. Auxiliary classes, however, are coupled with the main class to delegate the execution to it.

—**Reusability:** Code can be written within a single class, making it easier to refactor and to create more concise and reusable methods.

5. SOLUTION

The Secretary pattern proposes a way to segregate behaviors of a class with a complex interface in order to decrease the coupling of clients, while keeping the cohesion for code reuse. For each behavior, the pattern determines the creation of a **Secretary** class that represents it. Secretaries provide a narrowed view of the main class so that clients can be tied only to the functionalities that they really need.

All data that is used only by one behavior can be moved to the corresponding secretary class. The code, however, must be kept in the main class, now called **Boss**. Secretaries have methods that match and delegate to the methods (of the boss) that implement the behavior they represent. Nevertheless, they can have auxiliary methods that support the implementation of this behavior (such as methods that collect and give access to the data stored in them). All calls from the client to the boss are made indirectly through the secretaries.

In the example presented in section 2, it is possible to model each category of weapon as a boss while the *Shooting*, *Targeting* and *Reloading* behaviors get a secretary. Each subsystem of the game is a client and use only the secretaries it needs. This way, common code (such as specific collision algorithms) can be reused inside the boss while the subsystems loose their dependencies with the weapon classes. Settings that globally define each weapon instance (such as the maximum range) are kept in a single boss object while parameters that affect a single behavior (such as the number of remaining bullets) are kept in the secretaries (one for each player).

6. STRUCTURE

The SECRETARY pattern has two categories of classes:

—**Boss**: Main class with multiple behaviors. It keeps all the code implementation of the behaviors (spread through its methods) and holds data shared among them.

—**Secretary**: Auxiliary class used by clients. It provides access to a single behavior of the boss (forwarding the calls to its boss' instance) can store and collect data used only by this behavior.

These elements and their relations are illustrated by Figure 1. The class diagram presents a `Boss` class with two behaviors:

Behavior A. implemented by `methodA1` and `methodA2`; and

Behavior B. implemented by `methodB1` and `methodB2`.

These behaviors are represented by `SecretaryA` and `SecretaryB`, respectively. In the secretaries, the methods of the behaviors forward all arguments received to their equivalents in the boss. Additionally, `SecretaryB` passes a reference to itself as the first parameter of the call. This step is necessary only because the boss needs to access it to get the data. In `SecretaryA`, this parameter is unnecessary.

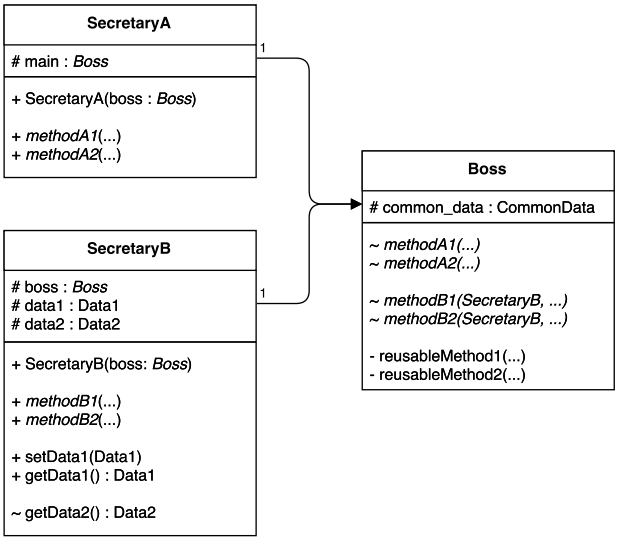

Fig. 1. Class diagram of the SECRETARY pattern:

One important aspect that is not explicitly represented by the diagram is the creation of the secretaries. The developer can either use a creational pattern (such as a FACTORY METHOD in the boss or an ABSTRACT FACTORY [Gamma et al. 1995]) to bind the secretary with a boss or she can make this binding directly by passing a reference to the boss in the secretary constructor.

7. DYNAMICS

The dynamics of the SECRETARY pattern are illustrated by Figure 2. Following the example presented in the last section, a `Client` uses part of the functionalities provided by a `Boss` through a `SecretaryB`. The client first executes `methodB1`, which does not store data, and then calls `methodB2`, which does. The secretary forwards requests passing itself as the first parameter of the calls, thus allowing the boss to access the data inside it.

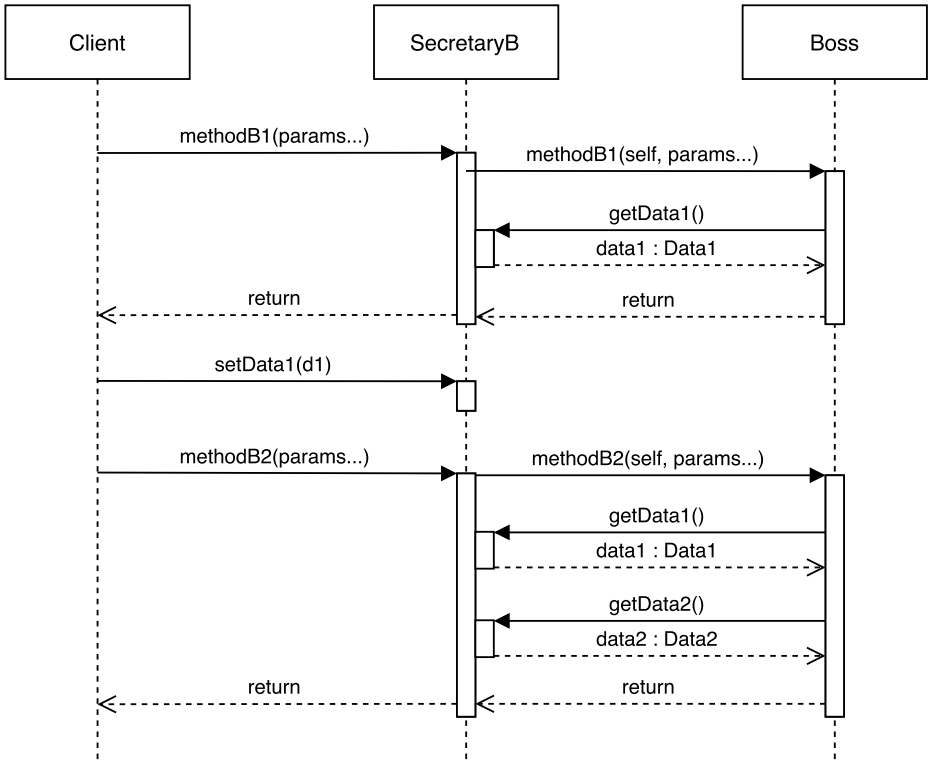

Fig. 2. Sequence diagram of the Secretary pattern:

8. APPLICABILITY

A class is a good candidate for the application of the Secretary pattern if it has two characteristics:

—**Multiple behaviors:** The candidate class has a complex interface in which part of its methods can be logically grouped in semi-independent subsets.

—**Code reuse:** The candidate class shares code among different behaviors.

Additionally, a good hint that the candidate class is suitable for the application of this pattern (thought not required) is that some of its attributes are not shared between different behaviors. This way, clients do not need to re-instantiate possibly heavy bosses, creating only light secretaries.

9. IMPLEMENTATION

The Secretary pattern can be implemented as follows: for each behavior in a class with multiple behaviors, create a secretary class that represents it. Add methods to the secretary that have the same signature of the methods of the boss that implement the behavior represented by this secretary. Implement these methods by forwarding all arguments to the boss. If possible, move all data used only by a single behavior to its secretaries. Add getters and setters to this data as needed. To access the data in the boss, pass a reference to secretary's instance (with other forwarded arguments) when delegating to the boss. In order to do it, add an extra parameter (the reference to the secretary's instance) in the affected methods of the boss. This last step is necessary only if the secretary will store data that need to be accessed by the boss.

```
render = RenderComponent.new
gameplay = GameplayComponent.new

hg = Handgun.new

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

render.render_path(targeter, direction)
render.menu(reloader)

shooter = hg.shooter(:silver_bullet)

gameplay.hit(shooter)
```

```
int main() {
  RenderComponent render;
  GameplayComponent gameplay;

  Handgun hg;

  auto targeter = hg.targeter();
  auto reloader = hg.reloader(12);
  reloader->reload(4);

  render.render_path(targeter, direction);
  render.menu(reloader);

  auto shooter  = hg.shooter(Projectile::silver);

  gameplay.hit(shooter);
}
```

Lst. 1. **Usage of the boss and secretary classes, in Ruby and C++**. This example shows a simple FPS game with a single kind of weapon: handguns. The game also has two subsystems: one for rendering and other to gameplay.The handgun class is implemented as a boss with three secretaries: shooters, responsible for finding the objects in the scenario that will be damaged by the weapon; targeters, responsible to calculate the expected path of the projectiles; and reloaders, whose aim is to store the number of projectiles available in a handgun.

Given that the delegating methods of a secretary replicate the signature of some methods of the boss, it is possible to use metaprogramming to automate the creation of these methods in a secretary. In C++, it is possible to use templates for this. In dynamic languages like Ruby, the same result can be achieved with reflection. The following listings show a Ruby and C++ implementation of the example presented in section 2. Listing 1 shows how clients (the rendering and gameplay subsystems) can use a boss (handgun class) and its secretaries (shooting, targeting and reloading) to make their work. Listing 2 shows the implementation of the bosses while Listing 3 shows the implementation of the secretaries. At last, Listing 4 presents helper classes and macros that implement some metaprogramming techniques that make it easier generate secretaries and their methods automatically. These helpers could be distributed as a library, making the adoption of the pattern easier.

10. CONSEQUENCES

This pattern makes refactoring easier, given that Secretary does not require a third object to share common code (as Component and Role Object do, given they split behaviors in many classes). 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. Secretaries can also be used to represent behaviors with static methods. Clients can avoid to re-instantiate possibly heavy bosses, creating only light secretaries. When adding a method to a behavior, not all clients need not be recompiled: only the ones that use the secretary that was changed. This pattern generates a design that follows the Interface Segregation Principle [Martin 2000], making clients do not depend on methods they do not use.

In order to use this pattern, developers need to handle more code. Particularly, there is a duplication of signatures between the methods in the boss and in its secretaries (although this duplication can be automated with metaprogramming).

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

  def path(targeter, direction)
    raytrace(direction).points
  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++**. In both languages, the handgun class inherit from an auxiliary class. In Ruby, this auxiliary class works for any example. It provides a helper method `secretary`, which allows the automatic creation of a secretary class by specifying its name (first argument) and the list of methods that will be accessed through it (remaining parameters). Additionally, it also creates a factory method [Gamma et al. 1995] to create secretaries from a boss instance. In C++, the auxiliary class works only for this example. It uses macros to implement factories methods for the three secretaries.

```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, ammunition_ = 0;
};

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

Lst. 3. **Secretaries definitions, in Ruby and C++**. Simple secretaries can be automatically generated with 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 secretaries. In Ruby, the auxiliary parent class `Secretary` provides the methods that delegate the calls to the boss. In C++, the macro `DELEGATE` is called once for each method. This is necessary because the language does not support looping through a list of methods. This macro is helpful because it hides the details of how to make the delegation.

```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. 4. **Metaprogramming, in Ruby and C++**. The details of the metaprogramming vary a lot according to the programming language. In Ruby, the static method `secretary` creates factory methods in the `Boss` and calls the static method `action` that injects, in `Secretary` classes, the methods that delegate to their boss. 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` (the second argument), naming the method as `name` (the last argument). 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 to delegate to the boss. `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 auxiliary base class used in this example. It provides a factory for all secretaries.

11. VARIATIONS

—Boss that keeps references to its secretaries
In the standard structure of this pattern, the relationship between the boss and the secretary is one way: a secretary keeps a reference to a single boss. However, the boss can keep references to the secretaries that serve it. This variation allows the boss to be used as a Repository [Evans 2004] of secretaries or to implement a creational pattern such as Singleton [Gamma et al. 1995].

—Immutable boss
In the standard structure of this pattern, there is no restrictions of mutability in the data stored in the boss and its secretaries. However, if the developer keeps all data inside the boss immutable with respect to its behaviors, he can execute the methods of these behaviors in parallel with no need of synchronization.

—Hierarchy of bosses
In the standard structure of this pattern, clients reduce coupling with a single boss. However, if the boss is the root of a hierarchy, clients can use a single secretary class to access specific implementations of their represented behavior without knowing the types of the concrete classes of the hierarchy.

—Multiple implementations of a behavior
In the standard structure of this pattern, each secretary corresponds to a single behavior of the boss, delegating the execution to its implementation. However, if the boss has alternative implementations of the behavior, developers can create a hierarchy of secretaries such that each concrete secretary class delegates to one of these alternative implementations.

12. RELATED PATTERNS

—Proxy [Gamma et al. 1995]:
Proxies are auxiliary objects whose interface is the same of another object and represents it within a system. Their implementation can extend the behaviors implemented by the real subject. Secretaries offer a narrowed view of the boss, containing methods and possibly data related to only one behavior. A secretary can also delegate to its boss passing itself as a parameter of the call (which is useful to give the boss access to the data stored within it).

—State [Gamma et al. 1995]:
State's concrete classes encapsulate data and implementation that represent the expected behavior of a context in a given phase of a state machine. Clients do not interact directly with the state objects and do not manage their use. Secretaries keep data related to a behavior but delegate all actions that must be executed to their bosses, allowing them to access their internal data if needed. Clients access the behaviors of the boss using only the secretaries.

—Strategy [Gamma et al. 1995]:
Strategy's concrete classes define the implementation of a behavior of a context. Clients access this behavior only through the context, keeping the same context object while replacing the strategies at runtime. Secretaries allow the access to behaviors already implemented by their bosses. Clients access these behaviors only through the secretaries, replacing the secretary object to access different implementations of the same behavior at runtime.

—Flyweight [Gamma et al. 1995]:
Flyweight's instances store shared data. Clients provide additional data to execute their methods. The primary goal of this pattern is to allow clients to reduce the amount of memory allocated when using many objects. Boss instances store data shared among different behaviors. Secretaries provide access to the additional data necessary to the behavior they represent. The primary goal of the pattern is to allow clients to depend only of the behaviors of the boss they need to use.

—Component [Nystrom 2014]:
Components encapsulate different behaviors of a container class, whose instance is built by injecting component objects in its construction. Clients access the behaviors through the container, which delegates to its components to execute the requests. Secretaries offer a narrowed view of the behaviors of their bosses. Clients access the behaviors they need using the secretaries, which delegate to their bosses to execute the actions.

—Role Object [Bäumer et al. 1998]:
Component roles extend the interface of a component core. Clients use component roles dynamically plugged in a component core, keeping access to all methods available in the core, in addition to the methods defined in the role. Secretaries provide a narrowed view of the interface of their bosses. Clients use secretaries to avoid knowledge of methods of the boss they do not need, reducing their dependency with its complex interface.

—Type Object [Johnson and Woolf 1998]:
Type class instances store common data that would be replicated among instances of a main class. Type class instances act as different superclasses of the main class. There is no restriction of how the methods should be divided and executed between both kinds of classes. The focus of this pattern is to reduce the replication of data, avoid subclass explosion and allow the creation of new "subclasses" at runtime. Bosses and secretaries divide data between themselves in a similar way. However, secretaries provide a narrowed view of the interface of their bosses, representing behaviors of them. Methods in the secretaries must delegate to the boss, which may use the secretaries to retrieve data specific of its behaviors. The focus of this pattern is to reduce the dependency between clients and a class with a complex interface while allowing the reuse reuse and refactoring of code that remains in a single class.

13. KNOWN USES

—ToPS framework [Kashiwabara et al. 2013]:
The Secretary pattern was originally identified in the refactoring of **ToPS** (*Toolkit for Probabilistic Models of Sequences*), a framework that facilitates the manipulation and integration of probabilistic models. ToPS implements probabilistic models as a hierarchy of immutable bosses (a combination of two variations described at section 11) and their different behaviors (training, evaluation, labeling, etc.) as a hierarchy of secretaries (another variation described at section 11). Applications are responsible to manage those secretaries to implement useful features for the end-users. This pattern was essential to build a new architecture for the framework, allowing developers to avoid unnecessary recompilations, to reuse the complex algorithms that define probabilistic models and to avoid duplication of trained parameters that define these models (varying only auxiliary data used by the algorithms). The implementation of ToPS can be found at: `https://github.com/topsframework/tops`.

—Rust compiler [Blandm 2015]:
Rust compiler automatically generates trait objects, which are a simple case of secretaries. Trait objects keep no data inside them, given that the compiler cannot automatically determine if some data will be used only by a single secretary. Methods in a trait object can either delegate their execution to the boss or perform some specific task defined in the specification of the trait. The implementation can be found at: `https://github.com/rust-lang/rust/blob/1.14.0/src/libcore/raw.rs`

—Go compiler [Donovan and Kernighanm 2015]:
Go compiler implements interfaces in a similar way to Rust's traits. However, the automatically generated objects are simpler than Rust's given that the user cannot define methods that will not delegate to the boss. The implementation can be found at: `https://github.com/golang/go/blob/release-branch.go1.8/src/runtime/iface.go`

14. ACKNOWLEDGEMENTS

We would like to thank our shepherd – Maurício Aniche – and the participants of the Writers' Workshop "Barracas" at SugarLoaf PLoP 2016 – Antonio Terceiro, Alessandro Leite, Joseph Yoder, Pedro Martos, Ralph Johnson and Tiago Boldt Sousa – for their insightful comments that significantly improved this paper.

REFERENCES

Bäumer, D., Bäumer, D., Riehle, D., Siberski, W., and Wulf, M. 1998. The Role Object Pattern. *WASHINGTON UNIVERSITY DEPT. OF COMPUTER SCIENCE*. 10

Blandm, J. 2015. *The Rust Programming Language: Fast, Safe, and Beautiful*. O'Reilly Media, Inc. 10

Donovan, A. A. and Kernighanm, B. W. 2015. *The Go Programming Language* 1st Ed. Addison-Wesley Professional. 10

Evans, E. 2004. *Domain-driven design : tackling complexity in the heart of software*. Addison-Wesley. 9

Fowler, M. and Beck, K. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley. 2

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. 3, 6, 9

Johnson, R. and Woolf, B. 1998. *The Type Object pattern*. Addison-Wesley. 10

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. 10

Martin, R. 2000. Design principles and design patterns. *Object Mentor*. 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. 8

Nystrom, R. 2014. Game programming patterns. 2, 10

Stroustrup, B. 2013. *The C++ Programming Language, 4th Edition*. Addison-Wesley. 8

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). HILLSIDE 978-1-941652-05-3