Objects for States

By Adam Petersen

Originally captured in *Design Patterns* [1], Objects for States is described in close conjunction with the Singleton pattern. This article will explain why this is an unfortunate combination and investigate better alternatives for implementing the pattern in C++. The complete source code for each implementation approach is available from my homepage [7].

The Singleton Connection

Design Patterns includes the name Objects for States only as an alias and the pattern is probably better known for its primary name: State. I prefer the name Objects for States because it expresses both the intent and resulting structure in a much better way. After all, the main idea captured in the pattern is to represent each state as an object of its own.

Besides the naming issue everything starts just fine in the pattern description and nothing indicates that Singleton is about to enter the scene. Not even as *Design Patterns* discusses implementation issues concerning the lifetime of state-objects do they actually mention Singleton. Turn the page and suddenly the pattern appears in the sample code with each state implemented as Singleton. Later on *Design Patterns* officially relates the two patterns by concluding that "State objects are often Singletons" [1]. However true that statement may be, is it a good design decision?

The case against Mr Singleton

The Singleton pattern is on the verge of being officially demoted to anti-pattern status. In order to get the freshest insider information possible, I decided to carry out an interview with the subject himself. Mr Singleton surprised me with his honesty and introspective nature.

- "Mr Singleton, you have been accused of causing design damage [6] and of leading programmers to erroneous abstractions by masquerading your tendencies to global domination as a cool object-oriented solution. What are your feelings?"
- "I'm just an innocent pattern, I did nothing wrong. I feel truly misunderstood."
- "But your class diagram included in *Design Patterns* looks rather straightforward. It doesn't get simpler than that - one class only - how could anyone possibly misunderstand that?"
- "Well, that's the dilemma". He continues with a mystic look on his face: "I look simple but my true personality is rather complex, if I may put it that way."

I understand he has more to say on the subject. I'll see if we can get further.

- "Interesting! Care to elaborate?" It seems like he just waited for this opportunity. Mr Singleton immediately answers, not without a tone of pride in his voice:
- "Sure, first of all I'm hard to implement".
- "Yes, I'm aware of that. Most writings about you are actually descriptions of the problems you introduce. What springs to my mind is, hmm, well, no offence, discussions about killing Singletons [9], a subject which makes matters even worse. There's also all the multithreading issues with you involved [10]."
- "Yeah, right, but my implementation is the minor problem. Can you keep a secret?"
- "Sure", I reply crossing my fingers.
- "Hmm, I shouldn't really mention this, but *Design Patterns* are over-using me."
- "Wow! You mean that you are inappropriately used to implement other patterns?"
- "Yes, you may put it that way. I mean, part of my intent is to ensure that a class only has one instance. But if an object doesn't have any internal state, then what's the point of using me? If there isn't any true uniqueness constraint, why implement mechanisms for guaranteeing only one, single instance?"

Reflecting on the above dialogue I notice that it describes a common problem with many implementations using Objects for States. In most designs the state objects are stateless, yet many programmers, including my younger self, implement them as Singletons. Sounds like some serious tradeoffs are made. After all, I like to take a test-driven approach and writing unit tests with Singletons involved is a downright scary thought. Mr Singleton agrees:

- "It's sad, isn't it? You end up solving the solution. Not only does it mean writing unnecessary code and that's a true waste; worse is that I'm wrong from a design perspective too."

There it is! Implementing Objects for States using Singleton is, I quote once more, "wrong from a design perspective". He said it himself. The good news is that in this case a better design also means less code and less complexity. But before jumping into the details of why and how, let's leave Mr Singleton for a while and recap the details of Objects for States.

Objects for States Recap

Objects for States works by emulating a dynamic change of type and the parts to be exchanged are encapsulated in different states. A state transition simply means changing the pointer in the context from one of the concrete states to the other. Consider a simple, digital stop-watch. In its most basic version, it has two states: started and stopped. Applying Objects for States to such a stop-watch results in the structure shown in Figure 1.

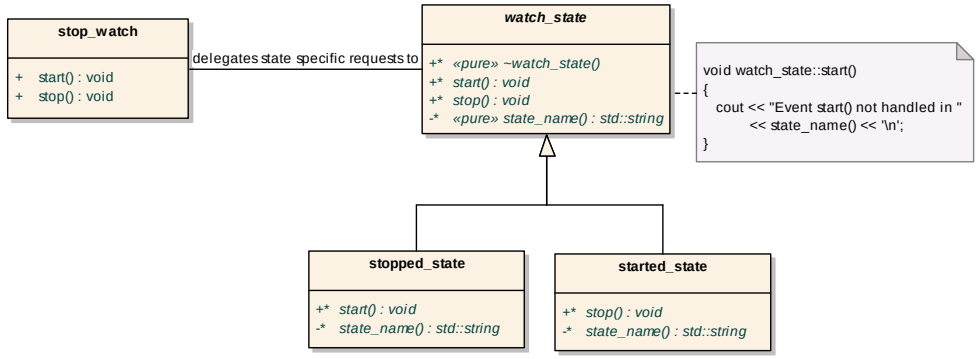

Figure 1: Structure of Objects for States

Before developing a concrete implementation, let's investigate the involved participants and their responsibilities:

- **`stop_watch`**: *Design Patterns* defines this as the context. The context has a pointer to one of our concrete states, without knowing exactly which one. It is the context that specifies the interface to the clients.
- **`watch_state`**: Defines the interface of the state machine, specifying all supported events. Depending upon the problem domain, **`watch_state`** may also implement default actions for different events. The default actions may range from throwing exceptions and logging to silently ignoring the events (the UML note in Figure 1 shows an example of a default action implemented in the **`start()`** function that sends a debug trace to standard output).
- **`stopped_state`** and **`started_state`**: These are concrete states and each one of them encapsulates the behaviour associated with the state it represents.

It depends

Design Patterns includes many examples of good OO designs. An example is its adherence to one of the most important design principles: "Programming to an interface, not an implementation". In fact all patterns in the catalogue, with one notable pathological exception - Singleton, adhere to this principle. Yet there are some subtle nuances to watch out for. Upon state transitions the pointer in the context has to be changed to the new state. The typical approach is to let each concrete state specify their successor state and trigger the transition. This way each state needs a link back to its context.

In its canonical form, Objects for States uses a friend declaration to allow states to access their context object. A friend declaration used this way breaks encapsulation, but that's not really the main problem; the problem is that it introduces a cyclic dependency between the context and the classes representing states. Such a dependency is an obstacle to unit tests and leads to big-bang integrations, although limited to the micro-universe of the context. Fortunately enough it is rather straightforward to break this dependency cycle. The first step is to introduce an interface to be used by the states:

```
class watch_state;

class watch_access
{
 public:
  virtual void change_state_to(watch_state* new_state) = 0;

 protected:
  ~watch_access() {}
};
```

This interface is realized in the context and each state is given a reference to it. A state can now make a transition by invoking **`change_state_to().`** Now, I deliberately didn't write exactly how the context shall implement the interface. From a design and usability perspective public inheritance isn't a good idea; **`watch_access`** is a result of our implementation efforts of weakening the dependencies and we really don't want to expose implementation details to clients of the **`stop_watch`**.

The perhaps simplest solution is offered by the idiom Private Interface [3]. All there is to it is to let **`stop_watch`** inherit **`watch_access`** privately. Now a conversion from **`stop_watch`** to **`watch_access`** is only allowed within the **`stop_watch`** itself. That is, the **`stop_watch`** can grant controlled access to its states and clients are shielded from the **`watch_access`** interface. Or are they really? Well, they are shielded from the conceptual overhead of the interface but there's more to it.

What worries me is that inheritance, private or not, puts strong compile-time dependencies upon the clients of **`stop_watch`**. In his classic book Effective C++, Scott Meyers advices us to "use private inheritance judiciously" [2]. Meyers also proposes an alternative that I find more attractive, albeit with increased complexity: declare a private nested class in the context and let this class inherit publicly. The context now uses composition to hold an instance of this class as illustrated in Figure 2. Not only is it cleaner with respect to encapsulation, it also allows us to control the compilation dependencies of our clients as it is possible to refactor it to a Pimpl [8] solution if needed.

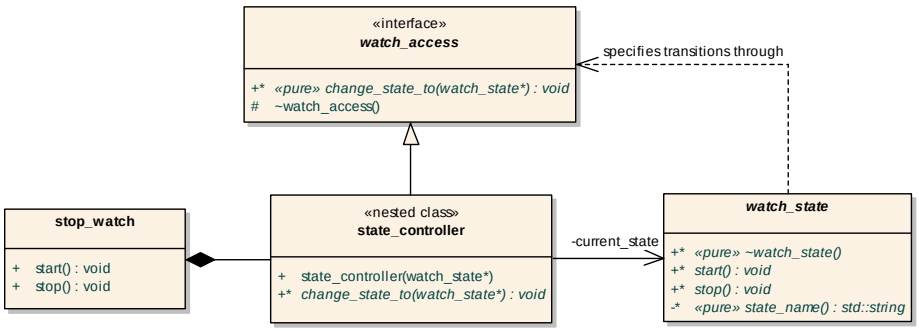

Figure 2: Decouple through a nested class

Enough of fancy diagrams – let's carve it out in code:

```
class stop_watch
{
 public:
  ...
 private:

  // Meyers Item 39: Prefer public inheritance plus composition in favour of private inheritance.
  class state_controller : public watch_access
  {
    ...
    public:
      ...
      virtual void change_state_to(watch_state* new_state)
      {
         ...
      }
    };

  state_controller state;
};
```

With the main structure of the context in place, we're ready to tackle the allocation of states.

A Dynamic Allocation Scheme

Our first approach is to allocate the states dynamically as they are needed. A state transition simply means allocating the new state, wrapped in a suitable smart pointer from boost [4], and passing it to the context. Here's an example on the stopped-state:

```
// watch_state.h
...
typedef boost::shared_ptr<watch_state> watch_state_ptr;

// stopped_state.cpp
void stopped_state::start(watch_access& watch)
{
   watch_state_ptr started(new started_state);

   watch.change_state_to(started);
}
```

The started-state has an identical mechanism, but of course it allocates **`stopped_state`** as its successor. With the allocation scheme in place we can implement the context:

```
// stop_watch.h
class stop_watch
{
 public:
  stop_watch();
  void start();
  void stop();

 private:

  class state_controller : public watch_access
  {
      watch_state_ptr current_state;

    public:
      state_controller(watch_state_ptr initial_state)
         : current_state(initial_state)
      {
      }

      // Hide the extra indirection for the client by using en masse delegation.
      watch_state_ptr operator->() const
      {
         return current_state;
      }

      virtual void change_state_to(watch_state_ptr new_state)
      {
         current_state = new_state;
      }
    };

  state_controller state;
};
```

Here we let the **stop_watch** specify its initial state upon construction:

```
// stop_watch.cpp
stop_watch::stop_watch()
  : state(watch_state_ptr(new stopped_state))
{
}
```

Our preference of public inheritance in combination with composition over private inheritance leads to an extra level of indirection. We can hide this indirection by overloading **operator->** in the **state_controller**, which makes the context's delegation to the states straightforward:

```
// stop_watch.cpp
void stop_watch::start()
{
  state->start(state);
}

void stop_watch::stop()
{
  state->stop(state);
}
```

Dynamic allocation of the states is a simple solution, yet it makes several tradeoffs:

+ Allows for stateful states, i.e. instance variables in the states.

- Potentially many and frequent heap allocations may have negative performance impact.

- Hard to change to sharing states (such a change ripples through all states).

- Dependent upon a concrete class (i.e. the next state), which is a barrier to unit tests.

Sharing States - The Return of the Singletons

With instance variables in the states, dynamic allocation is a simple solution. However, in most applications of Objects for States the state-objects are there just to provide a unique type and do not need any instance variables. *Design Patterns* describes this as "If State objects have no instance variables […] then contexts can share a State object" [1]. In their sample code *Design Patterns* notes that this is the case, only one instance of each state-object is required, and with that motivation makes each state a Singleton.

After my interview with Mr Singleton I promised to explain why this is the wrong abstraction. The reason is that the responsibility of managing state-instances is put on the wrong object, namely the state itself, and an object should better not assume anything about the context in which it is used. *Design Patterns* describes a particular case where only one instance is needed. This need, however, doesn't imply a uniqueness constraint on the state-objects themselves that would motivate the Singletons. Further, whether states should be shared or not should be decided in the context. Obviously the Singleton approach breaks this rule and, for all practical purposes, forces all states to be stateless.

To summarize, Singleton leads to:

1. an erroneous abstraction,
2. unnecessary code complexity,
3. superfluous uniqueness constraints,
4. and it seriously limits unit testing capabilities.

Clearly another approach would be preferable. However, before sharing any states, I would like to point to Joshua Kerievsky's advice that "it's always best to add state-sharing code after your users experience system delays and a profiler points you to the state-instantiation code as a prime bottleneck" [5].

Going Global

When implementing Objects for States the uniqueness constraint of Singleton is actually an unwanted by-product of the solution. So, let's focus on the second part of Singleton's intent: "provide a global point of access" [1]. These are the things programmers speak low about - nobody wants to get caught using globals, yet global variables are more honest about the intent than to camouflage them as Singletons. Consider the following code snippet:

```
// possible_states.h
class watch_state;

namespace possible_states{
extern watch_state* stopped;
extern watch_state* started;
}
```

```
// stopped_state.cpp
#include "possible_states.h"

void stopped_state::start(watch_access& watch)
{
   using possible_states::started;

   watch.change_state_to(started);
}
```

No constraints on the number of possible instances in the states themselves. But who defines them? The context seems like a good candidate:

```
// stop_watch.cpp
namespace{
stopped_state stopped_instance;
started_state started_instance;
}

namespace possible_states{
watch_state* stopped = &stopped_instance;
watch_state* started = &started_instance;
}
```

Except for the construction (we have to initialize our **`state_controller`** with **`possible_states::stopped`** instead of a dynamically allocated state), the rest of the context code stays the same. Any tradeoffs made? Yes, always. Here they are:

+ Conceptually simple and definitely simpler than the classic Singleton approach (same characteristics, but more honest in its intent).

+ No dependencies from the states upon concrete classes (only a forward declaration is actually used in `possible_states.h).`

+ Primitive but possible way to unit test individual states by use of link-time polymorphism (this technique uses the linker to link in different state definitions, i.e. test-stubs, instead of the real ones in `stop_watch.cpp`).

+- States are shared.

- Forced to share states, which makes it virtually impossible to use stateful states.

- Still not quite true to the "program to an interface" principle.

- Scalability problems with `possible_states.h`, which must be updated each time a state is added or removed.

In Control

Using link-time polymorphism to unit test? Yuck! Not particularly OO, is it? No, it sure isn't, but I wouldn't discard a solution just by that objection. Anyway, what about finally approaching a solution that removes the dependencies between the sub-states? Moving the state management into the **`state_controller`** makes it possible.

```
// watch_access.h
class watch_access
{
public:
   virtual void change_to_started() = 0;

   virtual void change_to_stopped() = 0;
 ...
};

// stop_watch.h
class stop_watch
{
  ...

  class state_controller : public watch_access
  {
      started_state started;
      stopped_state stopped;

      watch_state* current_state;

    public:
      state_controller()
         : current_state(&stopped)
      {
      }

      virtual void change_to_started()
      {
         current_state = &started;
      }

      virtual void change_to_stopped()
      {
         current_state = &stopped;
      }

      ...
    };
  ...
};
```

The **`state_controller`** allocates all possible states and switches between them as requested by the states. What's left to the states is specifying their successors in abstract terms:

```
// stopped_state.cpp
void stopped_state::start(watch_access& watch)
{
   watch.change_to_started();
}

// started_state.cpp
void started_state::stop(watch_access& watch)
{
   watch.change_to_stopped();
}
```

And here we are, finally programming to an interface and not an implementation. Let’s look at the resulting context:

+ The responsibility for the allocation scheme is where it should be: in the context.

+ States are easily shared among instances by making them static. Such a decision is taken in the context and not coded into the states themselves as in the traditional Singleton approach.

+ All states written towards an interface, which make them easy to unit test.

- Doesn't scale well. **`watch_access`** runs the risk of growing fat as it has to provide methods for all possible states, which is a similar problem to the global approach with `possible_states.h`.

A Generative Approach

The previous solution indicated potential scalability problems; adding new states requires modifications to **watch_access** and its implementer, **state_controller**. In my experience this has been an acceptable trade-off for most cases; as long as the state-machine is stable and relatively few states are used (5 - 10 unique states) I wouldn't think twice about it. However, in the ideal world, introducing a new state should only affect the states that need transitions to it. Reflecting upon our last example, although limited to only two states, the pattern is clear: the different methods for changing state (**change_to_started()**, **change_to_stopped()**) are identical except for the type encoded in the function name. Sounds like a clear candidate for compile-time polymorphism. The core idea is simple: each state instantiates a member function template with the next state as argument.

```
// Example from stopped_state.h
void stopped_state::start(watch_access& watch)
{
   watch.change_state_to<started_state>();
}
```

Each member function template instantiation creates the new state object and changes the reference in the context. Something along the lines of:

```
class X
{
...
   template<class new_state>
   void change_state_to()
   {
      watch_state_ptr created_state(new new_state);

      current_state = created_state;
   }
};
```

A quick quiz: in the listing above, what class should **X** be? The states specify their transitions by invoking methods on **watch_access** and by means of the virtual function mechanism the call is dispatched to the context. Now, there's no such beast as virtual member function templates in C++. The solution is to intercept the call chain and capture the template argument in an, necessarily non-virtual, member function template, create the new state instance there and delegate to the context by a virtual function.

```
// watch_access.h
class watch_access
{
public:

   template<class new_state>
   void change_state_to()
   {
      watch_state_ptr created_state(new new_state);

      change_state_to(created_state);
   }

protected:
   ~watch_access() {}
   typedef boost::shared_ptr<watch_state> watch_state_ptr;
private:
   // Delegate the actual state management to the derived class through this method.
   virtual void change_state_to(watch_state_ptr new_state) = 0;
};

// stop_watch.h
class state_controller : public watch_access
{
   watch_state_ptr current_state;

 public:
   state_controller()
   {
      // Specify the initial state.
      watch_access::change_state_to<stopped_state>();
   }

   virtual void change_state_to(watch_state_ptr new_state)
   {
      current_state = new_state;
   }
   ...
};
```

Considering the tradeoffs shows that the one step forward in scalability pushed us back with respect to dependency management:

+ Scales well, no know-them-all class; the compiler generates code to instantiate states.
- The states depend upon concrete classes.

Recycling States

The last example brought us back to a dynamic allocation scheme. However, that knowledge is encapsulated within **watch_access** and we can easily switch to another allocation strategy. For example, in a single-threaded context static objects are a straightforward way to share states and avoid frequent allocations:

```
// watch_access.h
class watch_access
{
public:
   template<class new_state>
   void change_state_to()
   {
      static new_state created_state;

      change_state_to(&created_state);
   }
   ...
};
```

State objects can also be recycled by introducing a variation of the design pattern Flyweight [1]. In fact, *Design Patterns* links these two patterns together with its statement that "it's often best to implement State […] objects as flyweights". Does the claim hold true? Let's try it out and see.

First each object is associated with a unique key. The idea is, that the first time an object is requested from the flyweight factory, a look-up is performed. If an object with the requested key already exists a pointer to that object is returned. Otherwise the object is created, stored in the factory, and a pointer to the newly created object returned. The example below introduces a pool for state objects in a **flyweight_factory** using the unique type-name as key.

```
template<class flyweight>
class flyweight_factory
{
public:

   typedef boost::shared_ptr<flyweight> flyweight_ptr;

   template<class concrete_flyweight>
   flyweight_ptr get_flyweight()
   {
      const std::string key(typeid(concrete_flyweight).name());

      typename pool_type::const_iterator existing_flyweight(pool.find(key));

      if(pool.end() != existing_flyweight) {
         return existing_flyweight->second;
      }
      else {
         flyweight_ptr new_flyweight(new concrete_flyweight);

         const bool inserted = pool.insert(std::make_pair(key, new_flyweight)).second;
         assert(inserted);

         return new_flyweight;
      }
   }

private:
   typedef std::map<std::string, flyweight_ptr> pool_type;
   pool_type pool;
};
```

The flyweights are fetched from the instantiations of the member function template in **watch_access**.

```
class watch_access
{
   typedef flyweight_factory<watch_state> state_factory;
   state_factory factory;

public:

   template<class new_state>
   void change_state_to()
   {
      change_state_to(factory.get_flyweight<new_state>());
   }
   protected:
   ~watch_access() {}

   typedef state_factory::flyweight_ptr watch_state_ptr;

private:
   // Delegate the actual state management to the derived class through this method.
   virtual void change_state_to(watch_state_ptr new_state) = 0;
};
```

The **state_controller** stays as before because the internal protocol, **change_state_to(watch_state_ptr)**, is left untouched.

+ Scales well, no know-them-all class; the compiler generates code to instantiate states.

+ Allows for sharing states among all instances of **stop_watch** by making the **flyweight_factory** static in **watch_access**.

+ Generic **flyweight_factory** for all default-constructable types.

- The states depend upon concrete classes.

- Relatively high design complexity.

Conclusion

As this article has highlighted the problems inherent in a Singleton based Objects for States solution, it feels fair to let Mr Singleton get the final word. After all, if I was successful his career may suffer. Will the two patterns finally be separated?

- "I sure hope so", Mr Singleton answers, "Clearly there are better alternatives and if I ever get the opportunity I'm prepared to sacrifice my link in Objects for States in the name of good design."
- "That's a great attitude and I'm delighted you take it that way. Speaking of design, any particular solution you would recommend?"
- "I don't think you can put it that way. Like all design alternatives each one of them comes with its own set of tradeoffs, which must be carefully balanced depending on the problem at hand."

References

1. Gamma, Helm, Johnson & Vlissides, "Design Patterns", Addison-Wesley, 1995
2. Scott Meyers, "Effective C++ Third Edition", Addison-Wesley, 2005
3. James Newkirk, "Private interface", 1997, http://www.objectmentor.com/
4. www.boost.org
5. Joshua Kerievsky, "Refactoring to Patterns", Addison-Wesley, 2004
6. Mark Radford, "SINGLETON – The Anti-Pattern!", Overload 57
7. Complete source code for this article, www.adampetersen.se
8. Herb Sutter, "Exceptional C++", Addison-Wesley, 2000
   The Pimpl idiom was originally described by John Carolan as the "Cheshire Cat".
9. John Vlissides, "Pattern Hatching", Addison-Wesley, 1998
10. Meyers & Alexandrescu, "C++ and the Perils of Double-Checked Locking", 2004, http://www.aristeia.com/

Acknowledgements

I would like to thank Drago Krznaric, Alan Griffiths, Phil Bass, and Richard Blundell for their valuable feedback.