The Secret Partner Pattern

Revision 3a by Bill Trudell, July 23, 2001
Submitted to the Pattern Languages of Programs
Shepherd: Neil Harrison PC Member: Kyle Brown

Thumbnail

This paper describes the Secret Partner Pattern. A Secret Partner is "a partner whose membership in a partnership is kept secret from the public"[1]. This pattern models the close, yet secretive association between a Partner and a Secret Partner. Clients or Customers can only interact with the original Partner and they have no knowledge of the Secret Partner. While the Partner and Secret Partner have a very close relationship, the Secret Partner also has information that truly remains a secret, even from the Partner.

Problem

In the real world, there is a need for close relationships with some amount of discretion. For example, the relationship between two co-workers can be close, but divulging salary information is not always prudent. Consider the case where 2 team members, a senior and junior developer, have shared responsibilities on a project. By chance, the senior member learns that the junior developer has a higher salary. Unable to deal with the injustice, the relationship erodes, productivity suffers and team morale is adversely affected. Divulging or accidentally discovering sensitive information can be dangerous and should be mitigated if possible. Software solutions model and parallel real world situations; therefore the problem can be expressed in the following question, "How can two software entities have a very close association while allowing one of them to keep some information secret from the other party?" This problem is solved by the Secret Partner Pattern.

Forces

There are several challenges this pattern and its solution must address. The co-worker example will be taken a bit further. Suppose the senior developer serves as the point of contact for their shared project. He or she will mentor the junior developer, allowing that person to learn some new skills in a safe environment, where risks can be managed and visibility is minimal. The senior member interfaces with the customer, attends meetings, and helps resolve requirements.

One force to be resolved is the desire of the senior member to shield the junior developer from the customer, company politics, etc. The junior developer also wants to stay anonymous in order to focus on learning the new technology and implementing the solution without the worry of being unduly scrutinized.

Another force is the necessity for the senior and junior developer to communicate efficiently and effectively. Since only the senior developer is attending the customer meetings, he or she knows the project vision and motivations behind the requirements. Even though the requirements are documented, they still must be explained and understood by the junior developer.

Finally, the junior developer has some idea the senior member is disgruntled about his current salary. If the junior member shares his higher salary with the more experienced senior developer, he might lose the chance to learn a new skill, or affect the delivery schedule due to hard feelings. Therefore, the salary information is best kept secret in order to maintain the relationship.

[1]Merriam-Webster On-line Dictionary, http://www.webster.com/

Forces (continued):

The challenges presented are especially an issue in the C++ programming language because the constraints are friendship and class accessibility. Friendship is limited to specific functions or the whole class; there is no middle ground. The forces are still the same even if Java is used to program the solution. However, the constraints are different because Java does not explicitly support friendship and it has different accessibility rules. Features like package and interfaces could be used to resolve the forces described. This paper will solve the problem with C++; a Java solution is beyond the scope of this paper but would nevertheless be interesting and possibly easier.

If friendship is only granted for certain functions, the nature of the friendship is restricted by public, protected and private accessibility. For close associations, this can be too constraining, requiring more over-head when accessing private data, etc. One object would incur the overhead of an artificial protocol with the cooperating object and this seems unnecessary. Only using friend functions may also require more maintenance as the classes mature; every time a function is added, the issue of friendship must be addressed.

The other extreme is granting friendship to a whole class. In this scenario, the grantor has no privacy. The friend class has unrestricted access to the grantor because the accessibility levels of public, protected and private are not enforced on friend classes. This seems too liberal for some situations. The classes could have no secrets from each other and encapsulation is broken.

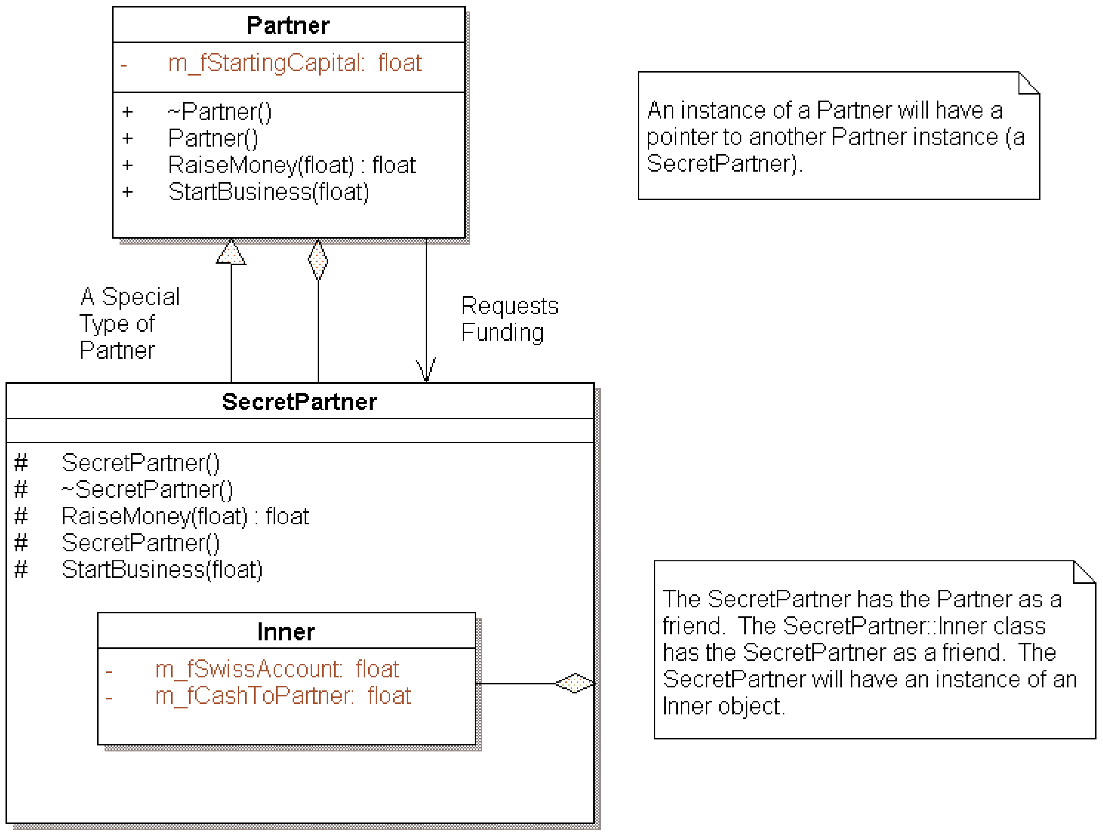
Solution

The declaration of an inner class is the key to limiting the scope of friendship between two C++ classes and will solve the problem of keeping some data truly secret. The Secret Partner Pattern balances intimacy and discretion by allowing one class to expose all of it's methods and only some of it's data to a collaborator or delegate.

The Webster definition of a Secret Partner is "a partner whose membership in a partnership is kept secret from the public"[2]. This pattern models the close, yet secretive association between a Partner and a Secret Partner. Clients or Customers can only interact with the Partner and they have no knowledge of the Secret Partner. While the Partner and Secret Partner have a very close relationship, the Secret Partner also has information that truly remains a secret, even from the Partner.

The Partner is implemented as a concrete base class. The Secret Partner derives from the Partner. The Secret Partner also declares an Inner class and contains an Inner class instance as shown in Figure 1. The example used in the solution is different than the one used to explain the problem. Here, a venture capitalist in the role of a Secret Partner is funding a company run by the Partner. This example conveys the pattern's intent but is not a solution that would be programmed in the real world.

Figure 1 – Secret Partner Pattern Class Diagram


Partner
- m_fStartingCapital: float
+ ~Partner()
+ Partner()
+ RaiseMoney(float) : float
+ StartBusiness(float)
An instance of a Partner will have a pointer to another Partner instance (a SecretPartner).
A Special Type of Partner
Requests Funding
SecretPartner
# SecretPartner()
# ~SecretPartner()
# RaiseMoney(float) : float
# SecretPartner()
# StartBusiness(float)
Inner
- m_fSwissAccount: float
- m_fCashToPartner: float
The SecretPartner has the Partner as a friend. The SecretPartner::Inner class has the SecretPartner as a friend. The SecretPartner will have an instance of an Inner object.

[2] Merriam-Webster On-line Dictionary, http://www.webster.com/

Solution (continued):

The header files for the Partner and Secret Partner classes are shown in Listings 1 and 2 respectively. Note that the Partner class header makes no mention of a Secret Partner instance. A typical software development kit would only include the Partner.h file and the compiled library with the appropriately exported functions. The kit would not include the SecretPartner.h since it's not necessary. Therefore, Application developers would not have immediate access or knowledge of the SecretPartner class.

Listing 1 – Partner Class header (Partner.h)

```
// Declarations for the Partner Class, Partner.h
#ifndef INCLUDED_PARTNER_H
#define INCLUDED_PARTNER_H

// Users or Application Developers only interact
// with the Partner.  (Apart from this comment,
// there is no mention of a SecretPartner, just a
// vague reference to another partner.)

class Partner
{
    public:
        Partner();
        virtual ~Partner();
        virtual void StartBusiness(float ventureCapital);
        virtual float RaiseMoney(float amountNeeded);
    protected:
        // none
    private:
        float m_fStartingCapital;
        Partner* m_pRef;    // pointer to another partner
};

#endif  // INCLUDED_PARTNER_H
```

Solution (continued):

The SecretPartner class declares an Inner class with private accessibility. The Inner class is now hidden from the Partner class even though it's a friend. However, the Inner class is also hidden from the SecretPartner; therefore the Inner class needs to make the SecretPartner its' friend. The Inner class declares its data as private. Now, the SecretPartner access its Swiss Bank account and keep it hidden from the Partner!

Listing 2 – Secret Partner header (SecretPartner.h)

```
// Declarations for the SecretPartner Classes
#ifndef INCLUDED_SECRETPARTNER_H
#define INCLUDED_SECRETPARTNER_H

#include "Partner.h"

// The SecretPartner is a special kind of Partner
// (it's hidden from the public and it has secret bank accounts)
class SecretPartner : public Partner
{
    public:
        // Declare Base Class Partner as a friend of the
        // SecretPartner so it can call the protected
        // constructor, or any other future methods, which
        // are hidden from users.
        friend class Partner;
    protected:
        SecretPartner();
        virtual float RaiseMoney(float amountRequested);
        virtual ~SecretPartner();
        virtual void StartBusiness(float ventureCapital);
    private:
        // No copying or assigning allowed for now
        SecretPartner(const SecretPartner&);
        const SecretPartner& operator=(const SecretPartner&);

        // Declare an Inner class in which the Secret Partner's
        // personal Data is hidden, even from the Partner who is
        // a friend and expecially from Users.
        //
        // The Inner Class must extend friendship to the outer,
        // so the outer has full access to the Inner.
        class Inner
        {
            friend class SecretPartner;
            private:
                float m_fCashToPartner;
                float m_fSwissAccount;
        };

        // Allocation of storage for the Inner
        Inner myInner;

};

#endif  // INCLUDED_SECRETPARTNER_H
```

Solution (continued):

The class definitions for the Partner and SecretPartner are shown in Listings 3 and 4. Since the SecretPartner derives from the Partner and an instance of a Partner allocates a SecretPartner, the construction process is messy. The Partner constructor initializes m_pRef to zero. Since the SecretPartner will also execute this constructor, an instance of the SecretPartner will have a zero for m_pRef. This is important later for safe destruction. The Partner instantiates a new SecretPartner in its StartBusiness function, setting the m_pRef variable. If the Partner had attempted to create the SecretPartner in its constructor, a race of executing the derived and base constructors would have occurred. (This is a good reason not to call virtual functions in a base class constructor, and a good reason not to perform heap allocation in a constructor.)

The destruction process is equally messy. When the Partner destructor is called for a Partner instance, the m_pRef will be non-zero. A delete is done on the pointer, which calls the destructor of the SecretPartner instance. Since the SecretPartner derived from the Partner and the destructors were declared virtual, the Partner destructor is called. This time, the m_pRef is zero and the function returns, allowing the SecretPartners destructor to complete and finally allowing the Partner instance to be destructed. Following this in the debugger was tedious.

Having said all that, implementing the SecretPartner as a singleton could simplify this confusion. The Partner would only call the _instance method, the SecretPartner would be responsible for the allocation and de-allocation of itself. However, it imposes an unnecessary restriction on the multiplicity between it and the Partner. It's a trade-off and the choice was made to minimize the restrictions on multiplicity.

Listing 3 – Partner Class Definitions (Partner.cpp)

```
// Definitions for the Partner Class

#include "Partner.h"
#include "SecretPartner.h"
#include <iostream>

using namespace std;

Partner::Partner()
{
    m_pRef = 0;
    m_fStartingCapital = 0;
}

void Partner::StartBusiness(float ventureCapital)
{
    m_fStartingCapital = ventureCapital;
    m_pRef = new SecretPartner;
    m_pRef->StartBusiness(ventureCapital);
}

float Partner::RaiseMoney(float amountNeeded)
{
    return ( m_pRef->RaiseMoney(amountNeeded) );
}

Partner::~Partner()
{
    delete m_pRef;
}

// End of Partner.cpp
```

<u>Solution (continued):</u>

The Partner is unable to access the SecretPartners Swiss Account data. It doesn't even know it's there really. Had the SecretPartner not wrapped that data in the inner class, the Partner would have been able to execute the statement *(static_cast<SecretPartner*>(m_pRef))->m_fSwissAccount*; to get that information. If the Partner executes the statement *(static_cast<SecretPartner*>(m_pRef))->myInner.m_fSwissAccount* , the compiler generates an error indicating that Partner cannot access a private member declared in the class 'SecretPartner::Inner', mission accomplished.

There's not much to be said about the SecretPartner implementation. The syntax for accessing the Inner class instance data is straightforward.

Listing 4 – SecretPartner Class Definitions (SecretPartner.cpp)

```
// Definitions for the SecretPartner Classes

#include "SecretPartner.h"

SecretPartner::SecretPartner() : Partner()
{
}

void SecretPartner::StartBusiness(float cashToPartner)
{
    myInner.m_fCashToPartner = cashToPartner;
    myInner.m_fCashToPartner = cashToPartner;

    // Secret Partner has 100 times the cash given to the
    // partner and it's stored in a Swiss account.
    myInner.m_fSwissAccount = 100.0F*cashToPartner;
}

float SecretPartner::RaiseMoney(float amountRequested)
{
    float amountGiven = 0.0F;

    // Don't give it all away, make sure to keep 10 Million,
    // it's not easy to live on less.
    if (( amountRequested <= myInner.m_fSwissAccount ) &&
        ( myInner.m_fSwissAccount >= 10000000.0F))
    {
        // You'll never get what you asked for from
        // a Venture Capitalist
        amountGiven = amountRequested * 0.9F;
        myInner.m_fSwissAccount -= amountGiven;
    }

    return amountGiven;
}

SecretPartner::~SecretPartner()
{
  // Nothing to do
}

// End of SecretPartner.cpp
```

Solution (continued):

A sample program for using this pattern is shown in Listing 5.

Listing 5 – Sample Program (Main.cpp)

```
// Main Program to Demonstrate the Secret Partner Pattern
#include "Partner.h"
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
{
    // A Partner is given 1 Million Dollars of Venture Capital
    // to start a New Company
    float startingCapital = 1000000.0F;
    Partner myPartner;

    myPartner.StartBusiness(startingCapital);

    cout << "Starting cash is " << startingCapital << endl;

    // At some point later, the company get's tight
    // for cash and needs another million dollars.
    // However, the public does not know about the Secret
    // Partner or where the money is going to come from.  The
    // employees can only hope and trust the Partner.

    float amtRequested = 1000000.0F;
    float amtReceived  = 0.0F;

    cout << "We need more money!" << endl;

    amtReceived = myPartner.RaiseMoney(amtRequested);
    if ( amtReceived < amtRequested )
    {
        cout << "We’re a little short, cut resources." << endl;
    }

    return 0;
}

// end of Main.cpp
```

Discussion

The solution presented addressed all stated challenges. The Partner, in its private section contained a pointer to another Partner, which was really the SecretPartner. Since the pointer is private, clients are not able to interact with it. Therefore, the Secret Partner is truly anonymous from the “public”. The Secret Partner granted the Partner class full friendship so there are no limitations on accessing the Secret Partners methods, most of which are protected, another safe-guard against accidental use by clients. The use of friendship is desirable for such a close association between 2 classes. Lastly, the Secret Partner was able to keep its Swiss Bank account hidden, even from the Partner.

The implementation actually has very few adverse side affects. One issue is the inheritance and aggregation of the Partner and SecretPartner that makes object construction and deconstruction difficult. An alternative implementation using the singleton was suggested. Another nuisance is the extra syntax required by the Secret Partner to access its Inner class instance data, which seems worth it in exchange for the safeguarding of confidential information.

Contraindications

The Secret Partner Pattern should not be used when friendship is not required; it would be better to use the public, protected and private accessibility levels to restrict access.

Related Patterns

There are several existing patterns related to the Secret Partner. It resembles the façade[3] pattern because clients interact with the Partner. In the example used for the solution, the Partner acted as the managing partner, yet the Secret Partner behaved like a Venture Capitalist who funded the company. It is similar to the envelope-letter idiom[4] where the Partner is the envelope and the Secret Partner is the letter inside. It is also similar to the adapter since clients interact with the Partner, but the SecretPartner does the real work.

Example Instances

In programming C++, streaming classes typically require friendship in order to fully implement the various operators like << and >>. Nevertheless, the class granting friendship may still want to guard certain attributes and behavior; after all it only needs to share its streaming nature. Another example might be the modeling of a tree node where the client interacts with a node in the role of the Partner, but the real implementation and node details are contained in the SecretPartner.

In real life, there are many similar examples. A puppet government, being sponsored and directed by a foreign super power might be an example.

References

The Author would like to recognize Ray Heath for suggesting the idea of limiting friendship and his encouragement for pursuing the ideas for this paper. Special thanks are extended to Neil Harrison for shepherding and helping to refine this paper. Finally, the Author would like to thank his wife Suzanne for her encouragement to pursue writing and sharing these ideas with the software community.

Authors Background:

The author has 18 years of diverse software development experience with a Bachelor of Science degree in Electrical Engineering. He has spent the last 9 years programming in C and C++ and is currently developing middleware for Capital One, a leading credit card issuer and financial Services Company. He has had papers published in Embedded Systems Programming[5] and the Journal of Object Oriented Programming[6].

[3] "Design Patterns", by Erich Gamma, et al., Addison-Wesley, 1995
[4] Coplien, J.O. "Advanced C++ Programming Styles and Idioms." Reading, MA. Addison-Wesley, 1992, pp. 316-323.
In addition, Cope recast the idioms as patterns in the following reference:
Coplien, J. O. "C++ Idioms Patterns." In Harrison, Neil B., Brian Foote, and Hans Rohnert, eds., *Pattern Languages of Program Design, Volume 4.* Reading MA. Addison-Wesley, 2000, pp. 167-198.
[5] Embedded Systems Programming, *Keys to Writing Efficient Embedded Code*, October 1997,CMP Media Inc. and *A Better Way To Process Messages*, May 2001,CMP Media Inc.
[6] JOOP, *The Access Proxy Pattern*, January 2001, 101 Communications