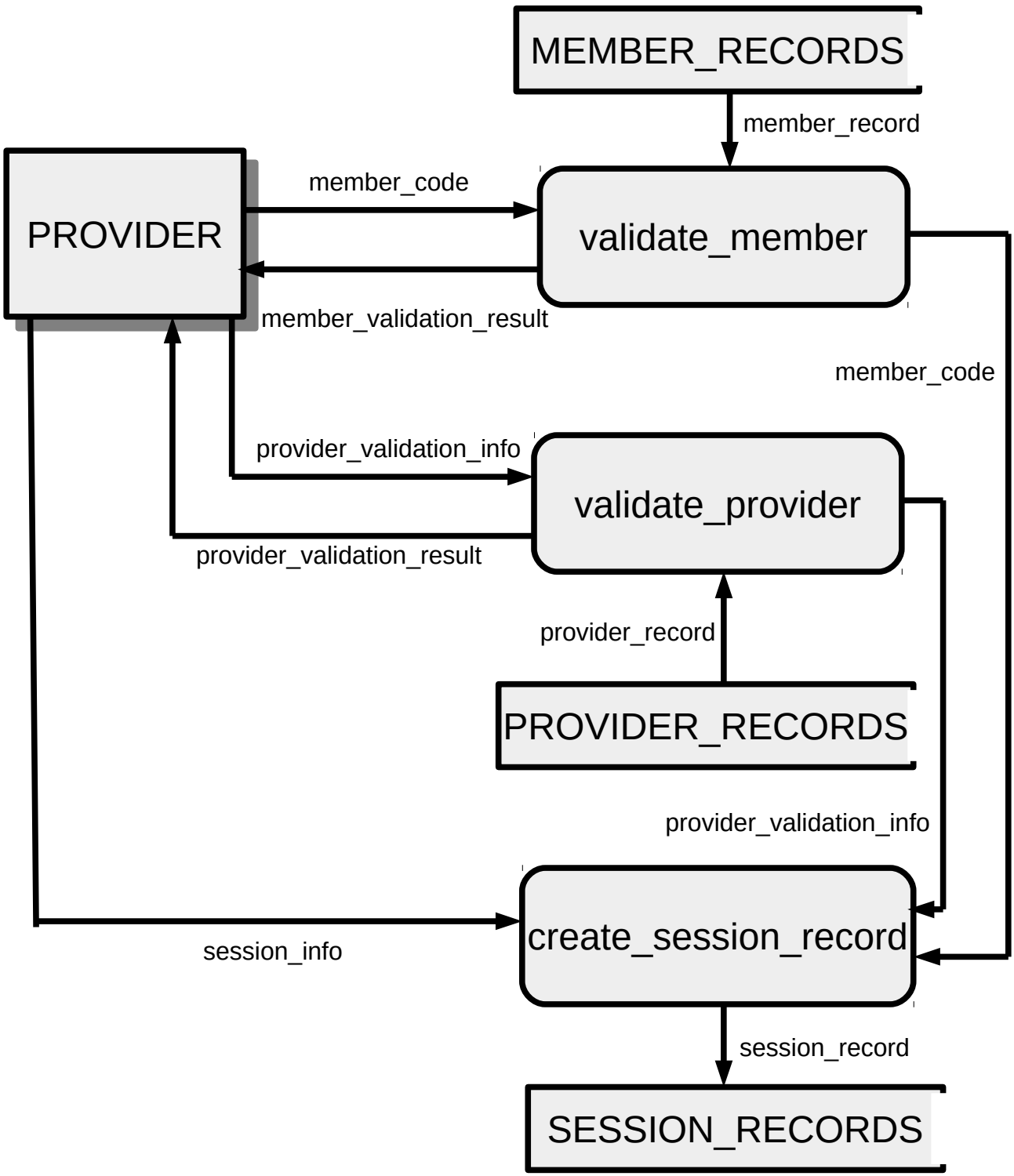
Comp 410 Case Study
Jason Bell 3078931 – May 27, 2015

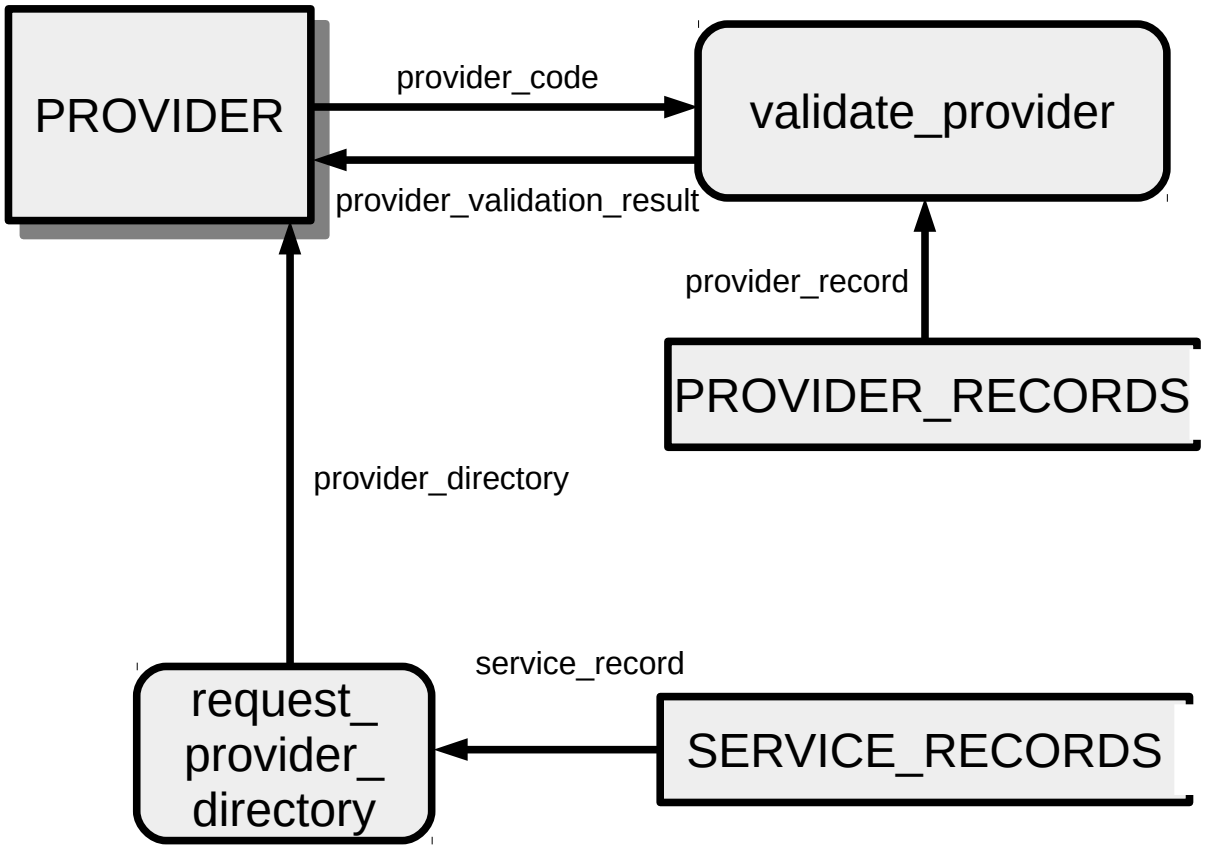
12.20 (Term Project) Using the technique specified by your instructor, draw up a specification document for the Chocoholics Anonymous product described in Appendix A.

Create data flow diagrams and data dictionaries using the requirements we defined in the previous assignment.

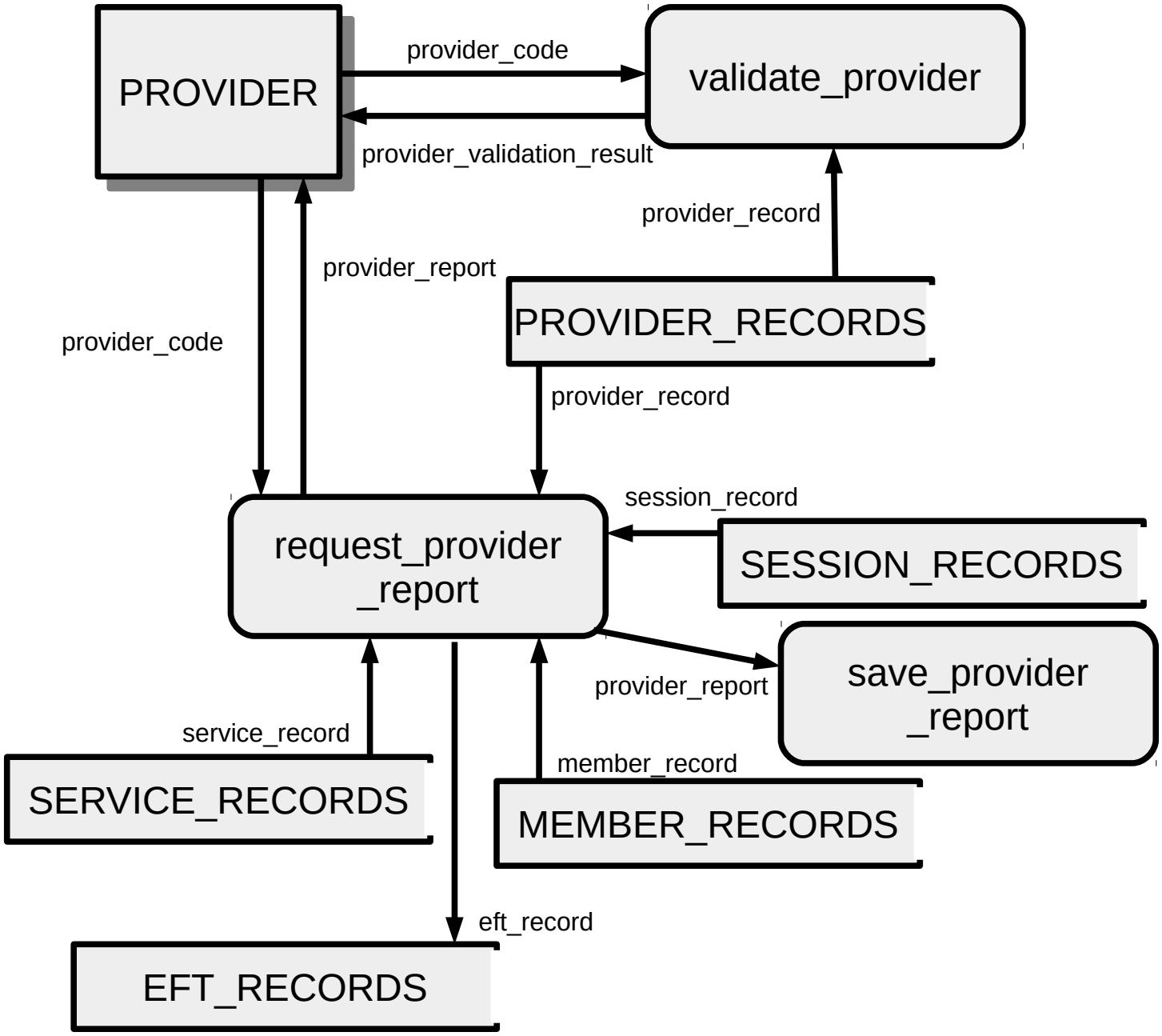
***Validate Member* and *Add Session* Uses Cases**



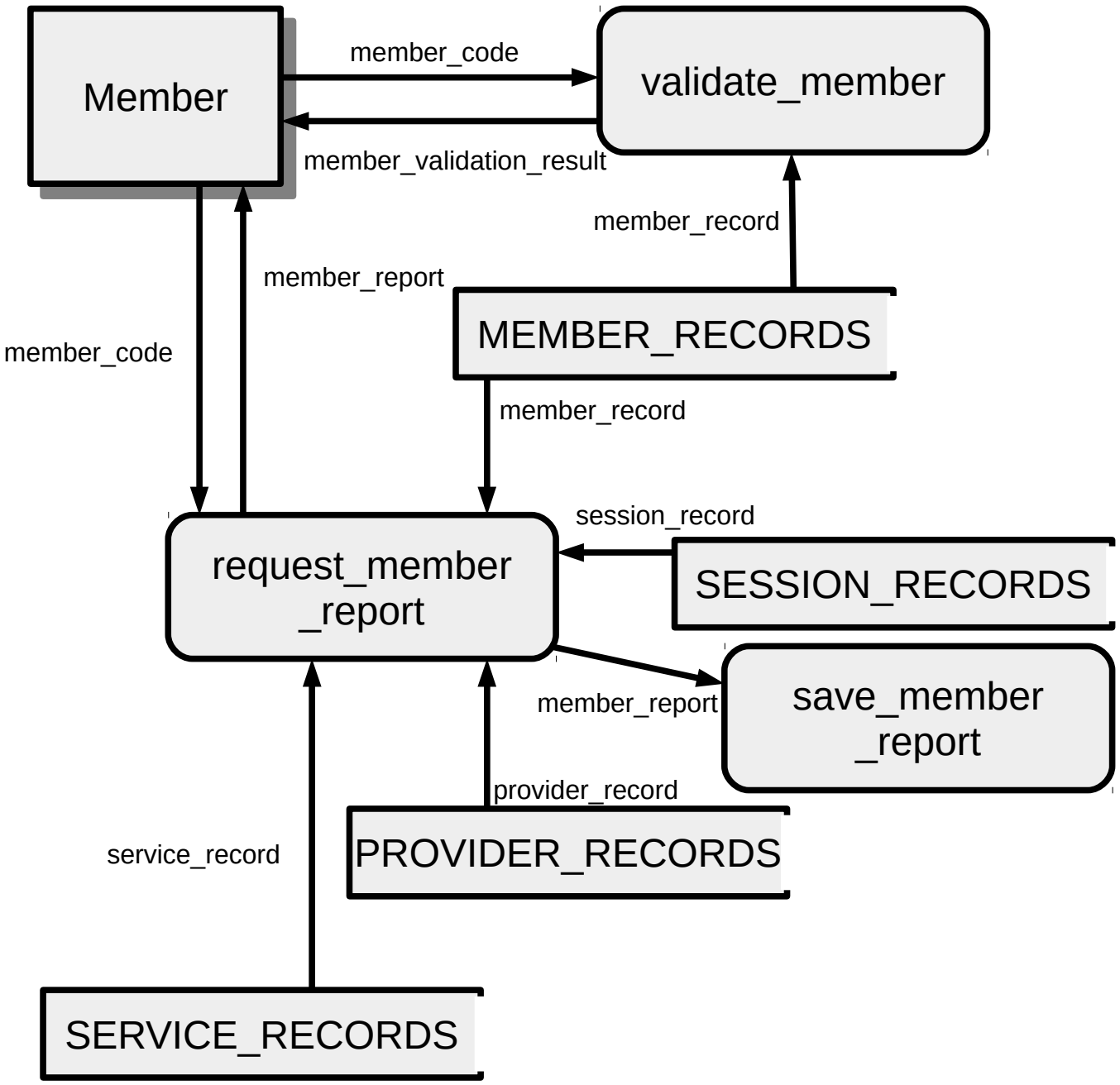
***Request Provider Directory* Use Case**



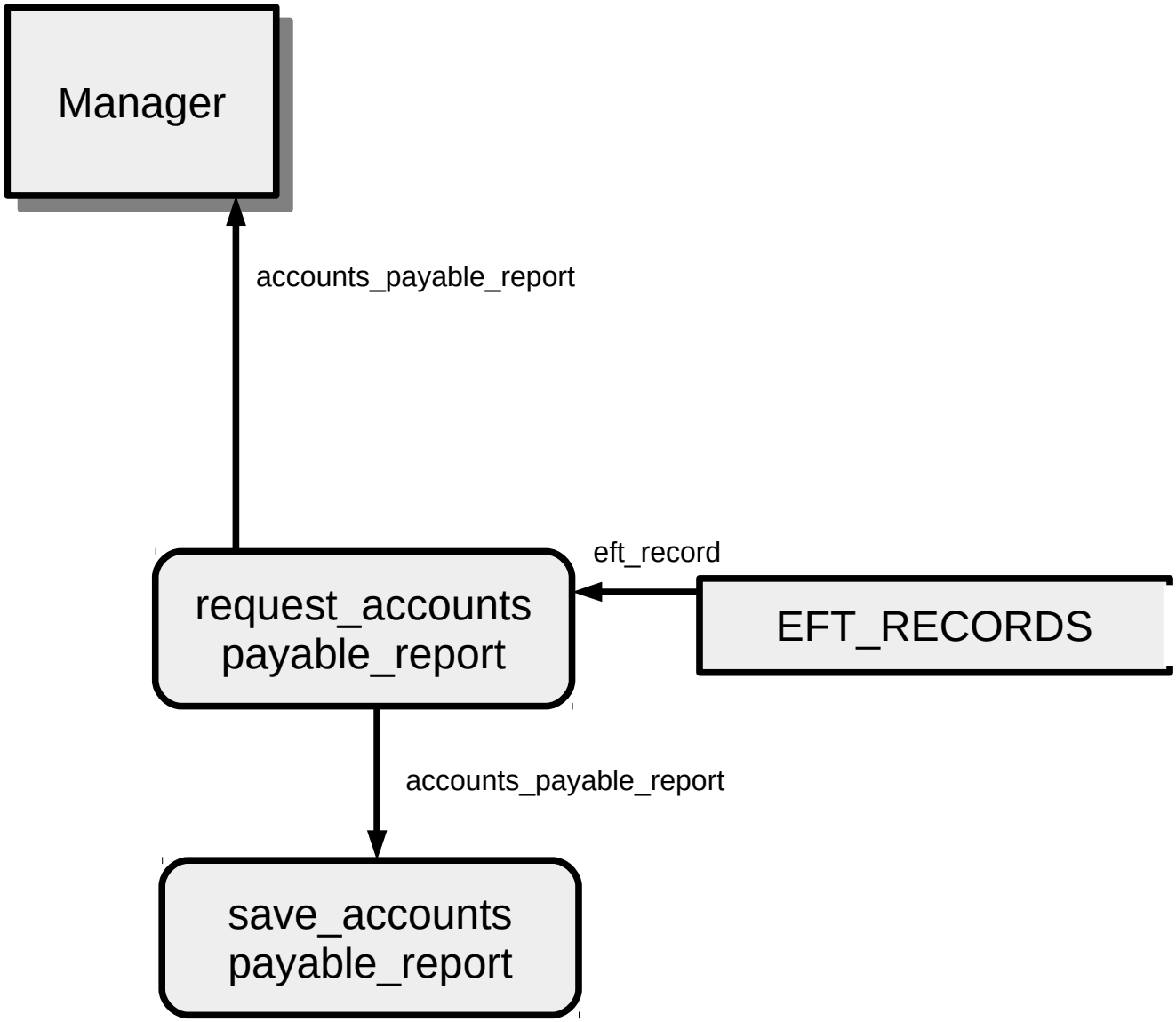
***Request Provider Report* Use Case**



***Request Member Report* Use Case**

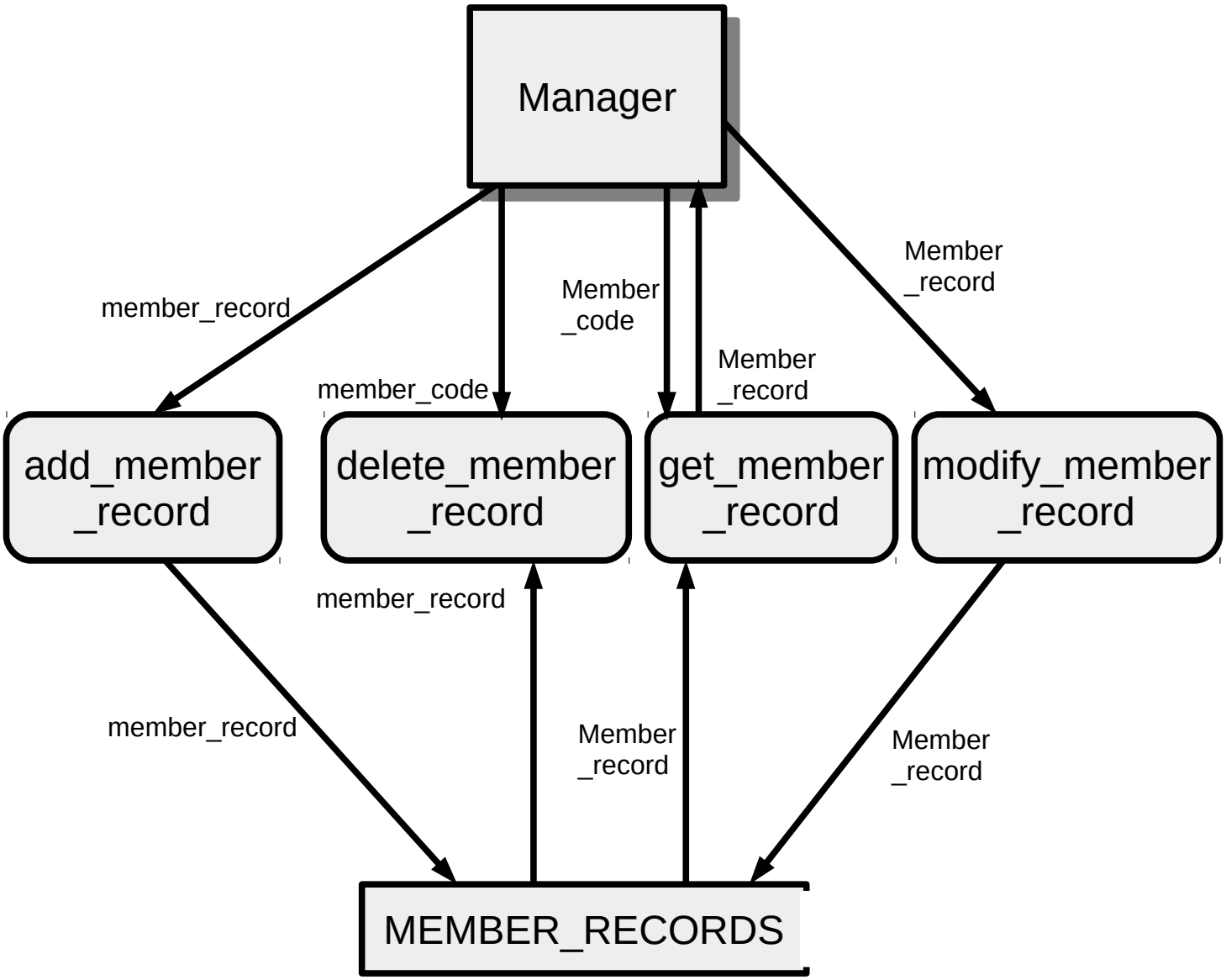


***Request Accounts Payable* Use Case**



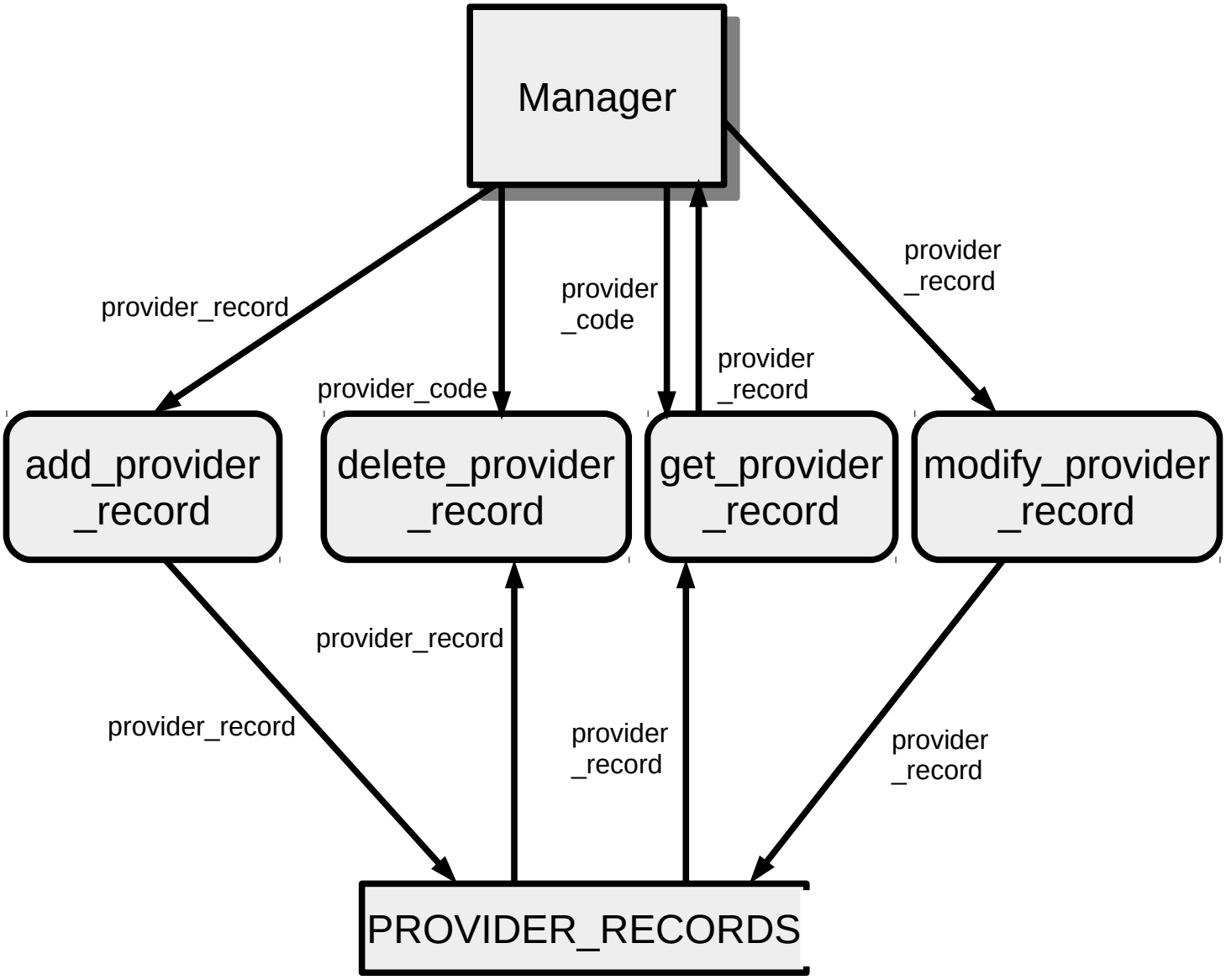
***Add, Delete, Get,* and *Modify Member* Use Cases**

Note the addition of the get use case: something that I didn't consider in the original modify use case. The terminal needs to get a copy of the member info before the manager can modify it! I suppose this outlines the importance of this analysis step.



***Add, Delete, Get*, and *Modify Provider* Use Cases**

Note the addition of the get use case: something that I didn't consider in the original modify use case. The terminal needs to get a copy of the provider info before the manager can modify it!



Data Dictionary

Name of Data Element	Description	Narrative
accounts_payable_report	See *Request Accounts Payable Report* use case.	List of all providers owed payment. For each provider owed money, show total money owed and number of sessions for that provider.
add_member_record	Procedure: Input parameter: member_record Output parameter: None	Accepts a new member record, fills out the member_code field of the record with a new and unique member code, and writes the record to the MEMBER_RECORDS data store.
add_provider_record	Procedure: Input parameter: provider_record Output parameter: None	Accepts a new provider record, fills out the provider_code field of the record with a new and unique provider code, and writes the record to the PROVIDER_RECORDS data store.
comments	100 character string	Optional comments for a sessions record.
create_session_record	Procedure: Input parameters: session_info provider_validation_info member_code Output parameters: session_record	This procedure accepts the data required to return a constructed session_record for storage in SESSION_RECORDS.
delete_member_record	Procedure: Input parameters: member_code Output parameters: None	This procedure retrieves from the MEMBER_RECORDS data store the record with a member code matching the given code. Upon retrieval, the record is stored elsewhere, then deleted from MEMBER_RECORDS.
delete_provider_record	Procedure: Input parameters: provider_code Output parameters: None	This procedure retrieves from the PROVIDER_RECORDS data store the record with a provider code matching the given code. Upon retrieval, the record is stored elsewhere, then deleted from PROVIDER_RECORDS.

eft_record	Record comprising fields provider_name provider_code total_fees	
get_member_record	Procedure: Input parameters: member_code Output parameters: member_record	This procedure accepts a valid member code, retrieves the record with this code from the MEMBER_RECORDS data store, and returns the record.
get_provider_record	Procedure: Input parameters: provider_code Output parameters: provider_record	This procedure accepts a valid provider code, retrieves the record with this code from the PROVIDER_RECORDS data store, and returns the record.
member_city	14 character string	The city of a member.
member_code	9 digit integer	The identifying code for a member.
member_email	25 character string	The e-mail address of a member.
member_name	25 character string	The full name of a member.
member_record	Record comprising fields member_code member_name member_street_address member_city member_state member_zip member_email	Storage format for member record data.
member_report	See *Request Member Report* use case.	A report summarizing services used by a member for the current week.
member_state	2 character string	The state of a member.
member_street_address	25 character string	The street address of the member.
member_validation_result	1 digit integer	Result of validation of a member code. 0 indicates validation, 1 indicates that the member code was not found, and 2 indicates that the member is suspended.
member_zip	5 digit integer	The ZIP code of a member.
modify_member_record	Procedure: Input parameter: member_record Output parameter:	This procedure accepts a member record with a valid, existing member code, and overwrites the record with that

	None	code found in MEMBER_RECORDS.
modify_provider_record	Procedure: Input parameter: provider_record Output parameter: None	This procedure accepts a provider record with a valid, existing provider code, and overwrites the record with that code found in PROVIDER_RECORDS.
provider_city	14 character string	The city of a provider.
provider_code	9 digit integer	The identifying code for a provider.
provider_directory	Record comprising fields list of service_records	A directory of services with their names, codes, and fees.
provider_email	25 character string	The email address of a provider.
provider_name	25 character string	The full name of a provider.
provider_record	Record comprising fields provider_name provider_code provider_street_address provider_city provider_state provider_zip provider_email	Storage format for provider record data.
provider_report	See *Request Provider Report* use case.	Report summarizing all services provided by a provider for the current week.
provider_state	2 character string	The state of a provider.
provider_street_address	25 character string	The street address of a provider.
provider_validation_info	Record comprising fields provider_code session_date	Used to validate that this provider exists. It's clumsy that the session date is included here, and not in session_info, but it's how the requirements are described.
provider_validation_result	1 digit integer	0 indicates validation, 1 indicates that the provider code was not found.
provider_zip	5 digit integer	The ZIP code of a provider.
request_accounts_payable_report	Procedure: Input parameter: None Output parameter:	Compiles and returns an accounts payable report for sessions for the current week.

	accounts_payable_report	
request_member_report	Procedure: Input parameter: member_code Output parameter: member_report	Compiles and returns a member report for the current week by querying all sessions for the week, then querying all other data stores for supplementary information. For example, session_code only stores the provider code, so PROVIDER_RECORD must be queried to get the name of the provider.
request_provider_directory	Procedure: Input parameter: none Output parameter: provider_directory	Compiles and returns a provider directory from all service records in the data store.
request_provider_report	Procedure: Input parameter: provider_code Output parameter: provider_report	Compiles and returns a provider report for the current week by querying all sessions for the week, then querying all other data stores for supplementary information. For example, session_code only stores the member code, so MEMBER_RECORD must be queried to get the name of the member. Each session is saved to file as an EFT record.
save_accounts_payable_report	Procedure: Input parameter: accounts_payable_report Output parameter: none	Saves a given accounts payable report to a file. File name details unclear at this time.
save_member_report	Procedure: Input parameter: member_report Output parameter: none	Saves to given member report to a file with a file name starting with the member name, followed by the date of the report.
save_provider_directory	Procedure: Input parameter: provider_directory Output parameter: none	Saves the given provider directory to a file with a file name starting with the provider name, followed by the date of the report.

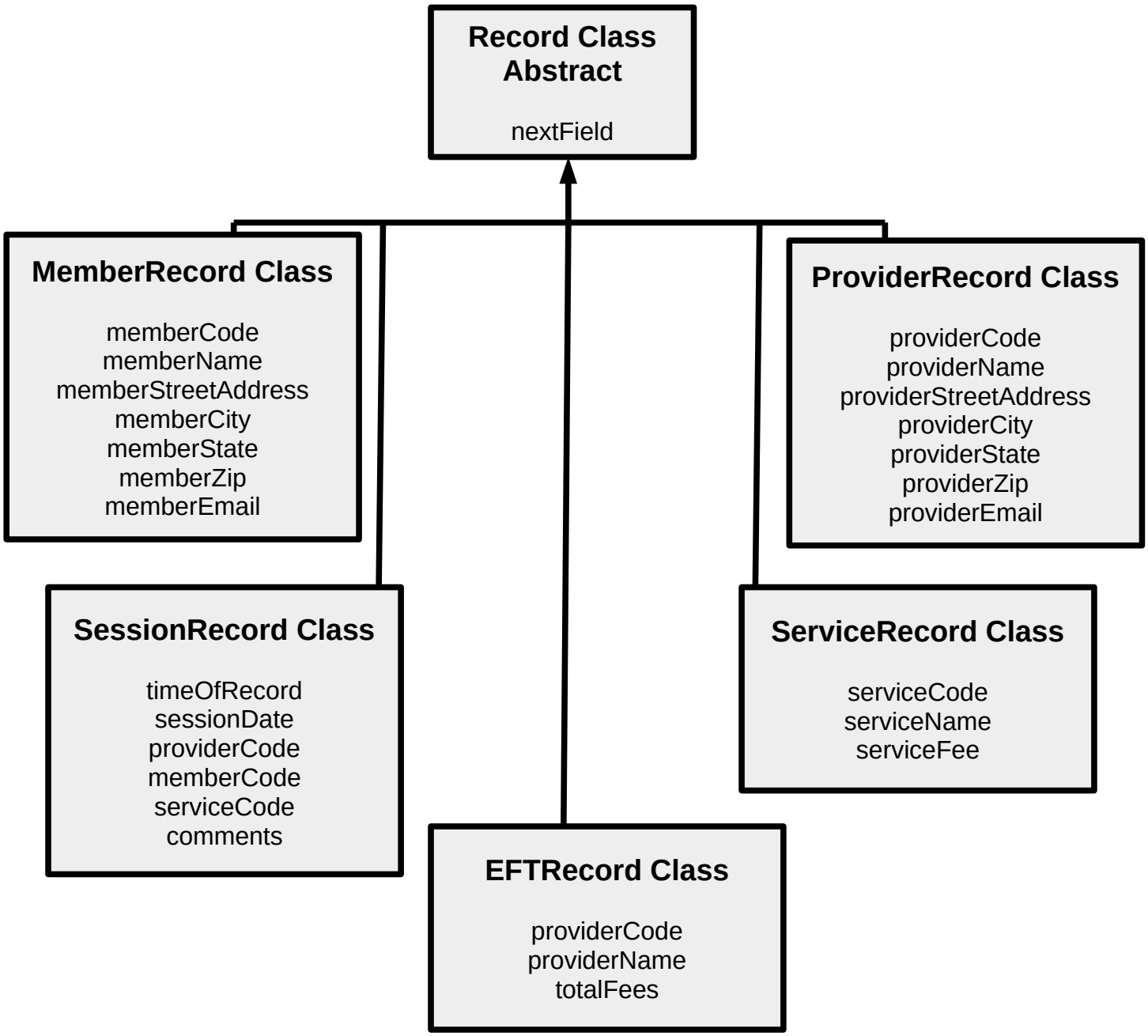
save_provider_report	Procedure: Input parameter: provider_report Output parameter: none	Saves the given provider report to a file. File name details unclear at this time.
service_fee	5 digit integer	The fee for a service.
service_name	25 character string	The name of a service.
service_code	6 digit integer	The service code for a particular provider service.
service_record	Record comprising fields service_code service_name service_fee	
session_date	10 character string	The date a session was performed as MM-DD-YYYY.
session_info	Record comprising fields time_of_record service_code comments	Information on the time, code, and extra comments for a session.
session_record	Record comprising fields time_of_record session_date provider_code member_code service_code comments	Storage format for session record data.
time_of_record	19 character string	A date and time in the format MM–DD–YYYY HH:MM:SS
validate_member	Procedure: Input parameter: member_code Output parameter: member_validation_result	This procedure accepts a member code and validates that a member record with that code exists. The result of this validation is returned as member_validation_result.
validate_provider	Procedure: Input parameter: provider_validation_info Output parameter: provider_validation_result	This procedure accepts provider validation info or a provider code and validates that a provider record with that code exists. The result is returned as provider_validation_result.

13.22 (Term Project) Perform the analysis workflow of the Chocoholics Anonymous product described in Appendix A.

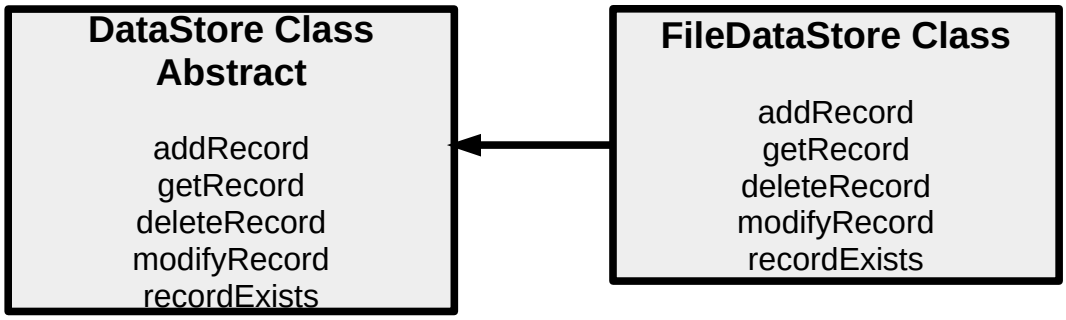
First, let's look at arguably the simplest use cases: those for adding, deleting, getting, and modifying records. I will ad corresponding methods for manipulating service records, as I realize that I overlooked this in the previous assignment (again, showing the importance of iterative/incremental methodologies).

As I suggested in the previous assignment, all records will derive from a common base case to assist re-usability and to minimize the number of ways in which records need to be written. That is, there will be one record writing instance that accepts the generic record. The system would have different instances of this class for each record type.

So, here are the record classes:

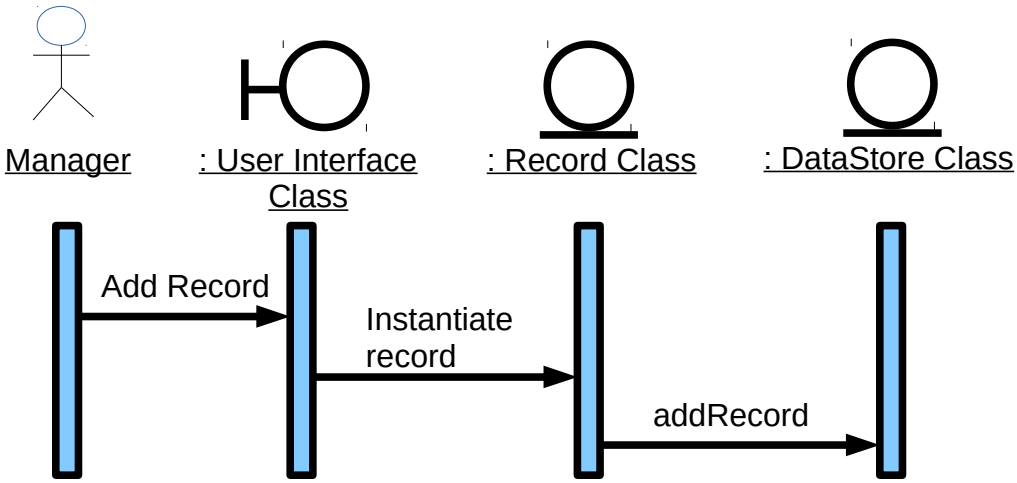


We can define an abstract data store class and a file based implementation thusly:

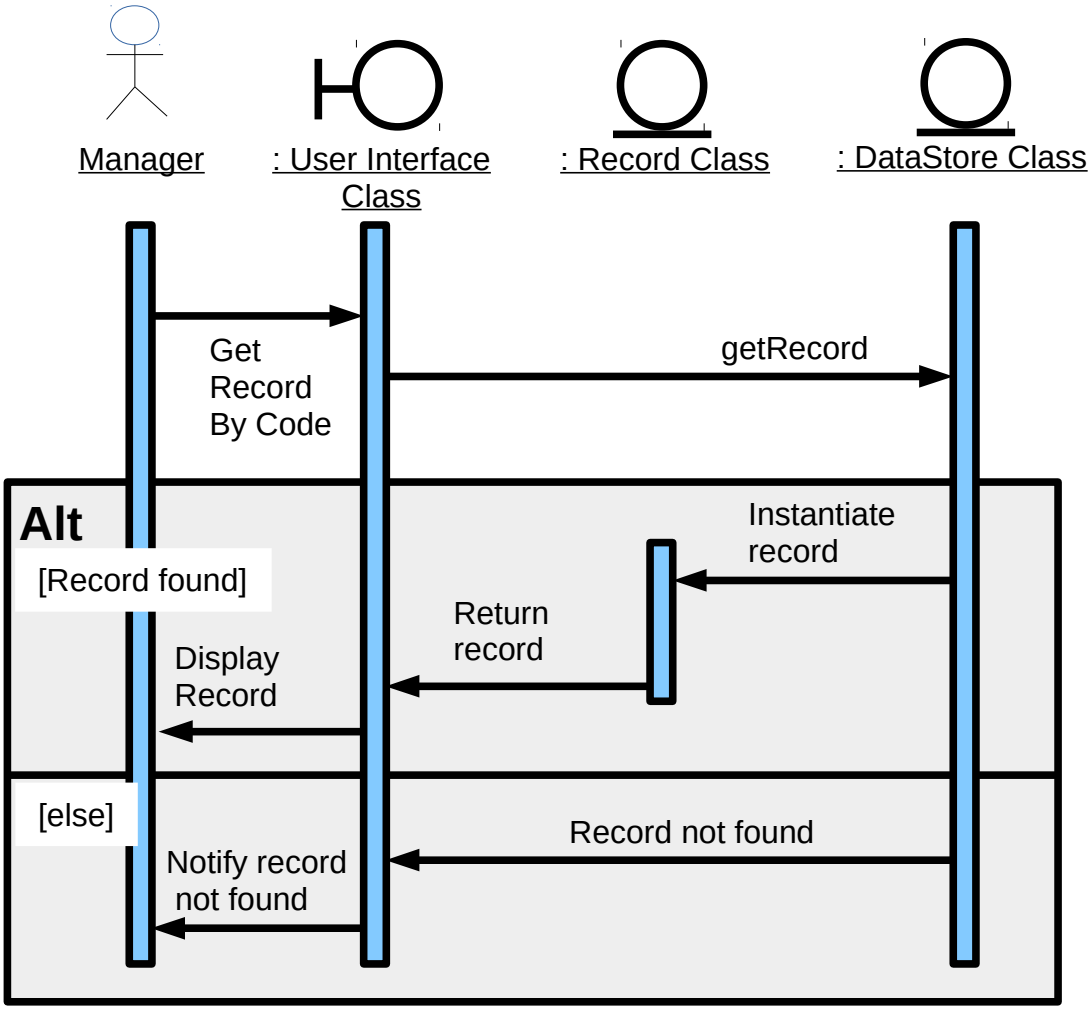


At present we can describe the following UML classes in a sequence diagram of the add/delete/get/modify record use cases. They are merged into one for brevity, since each will be the same, just with a different implementation of record.

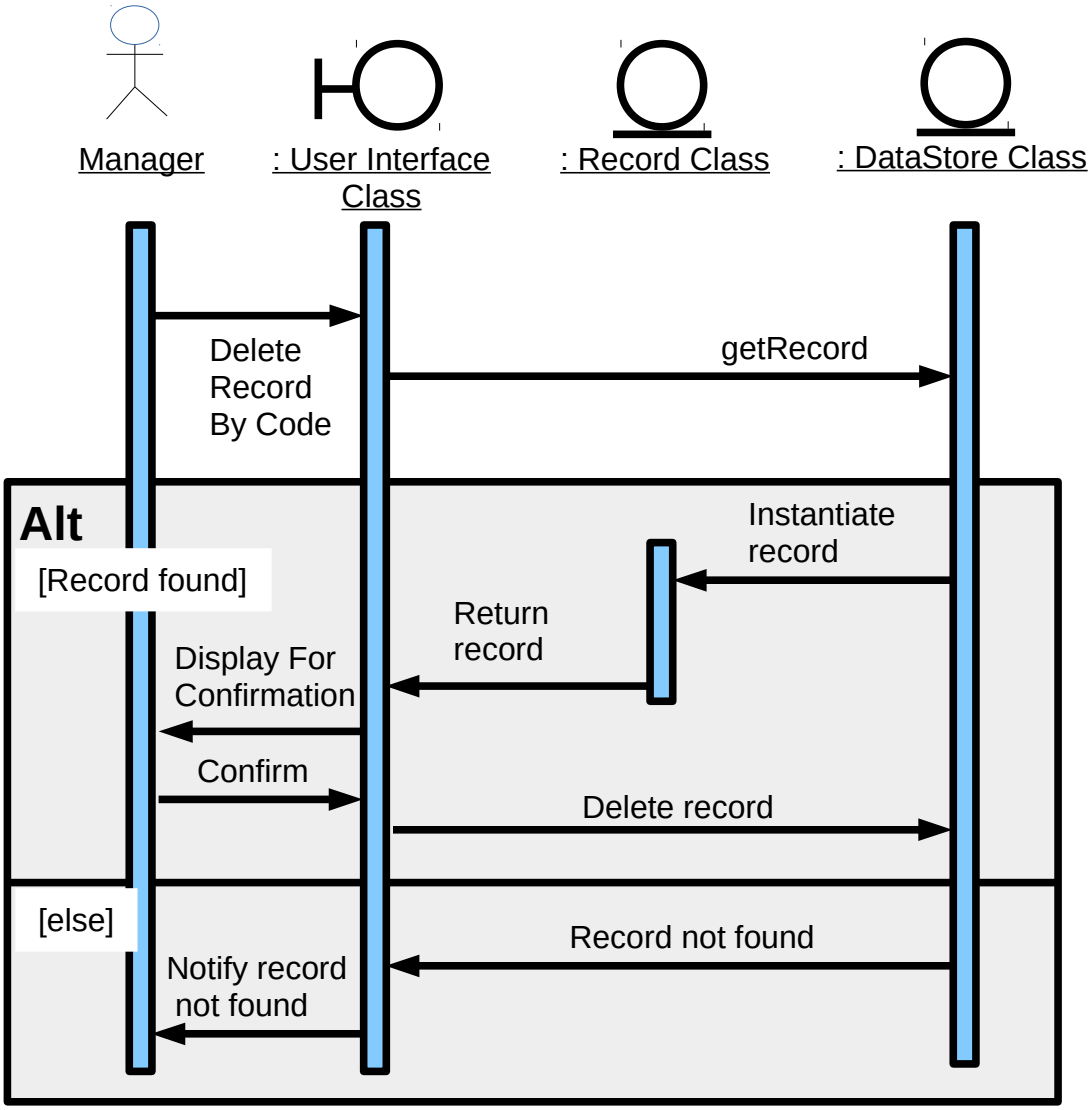
Starting with the simplest: the add new record use case.



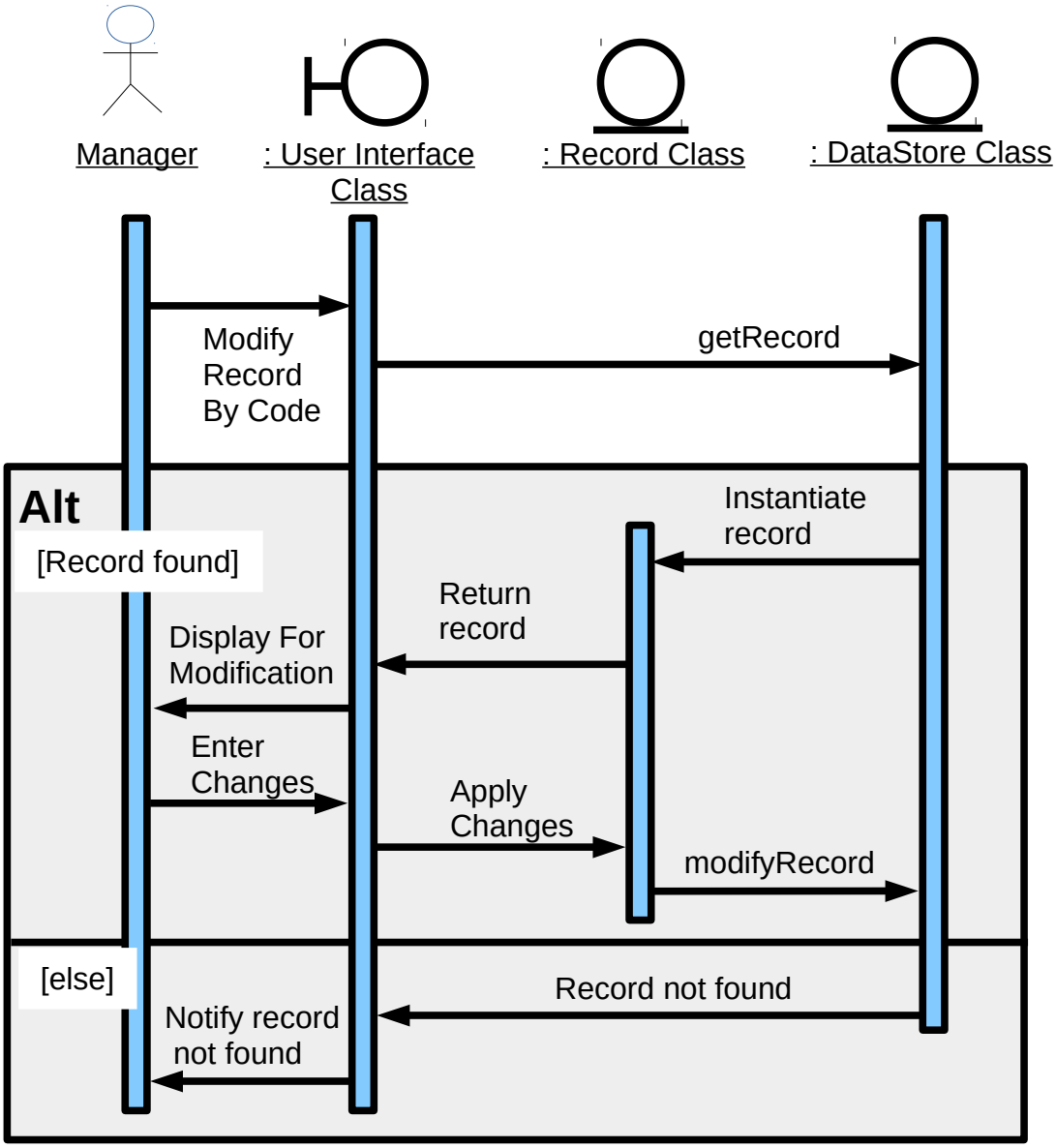
The get record use case:



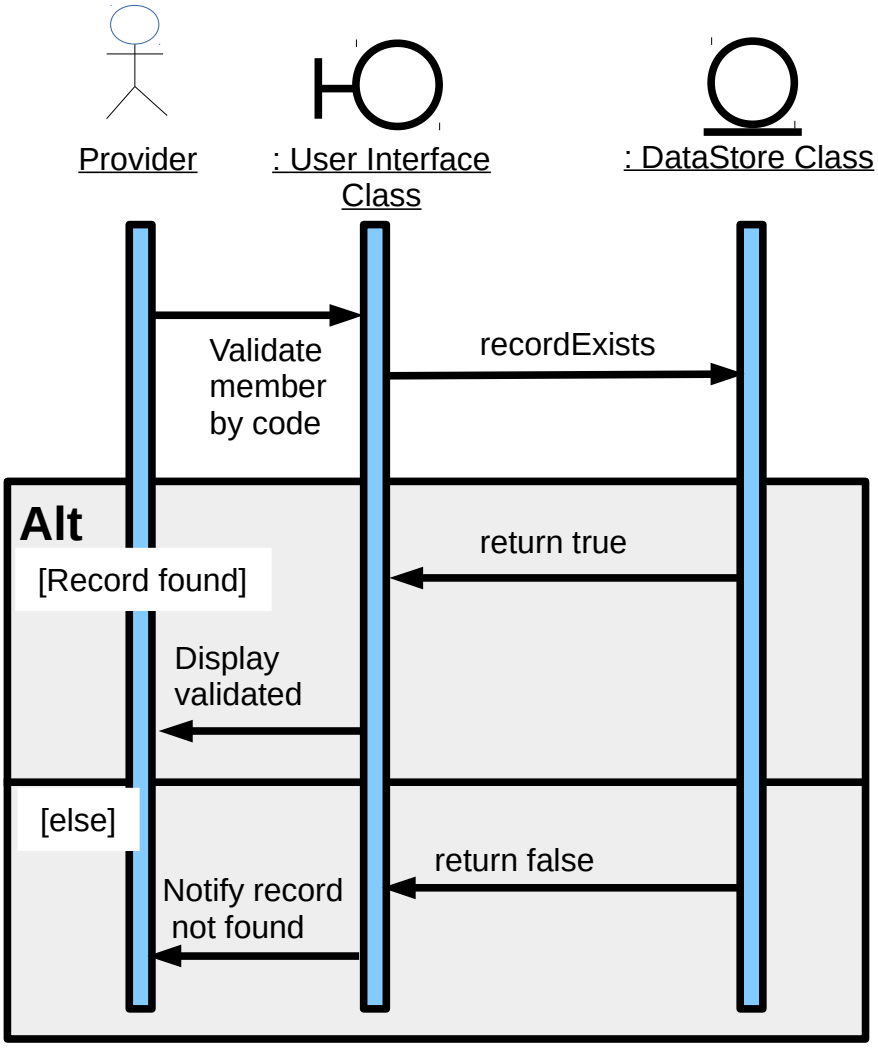
The delete record use case:



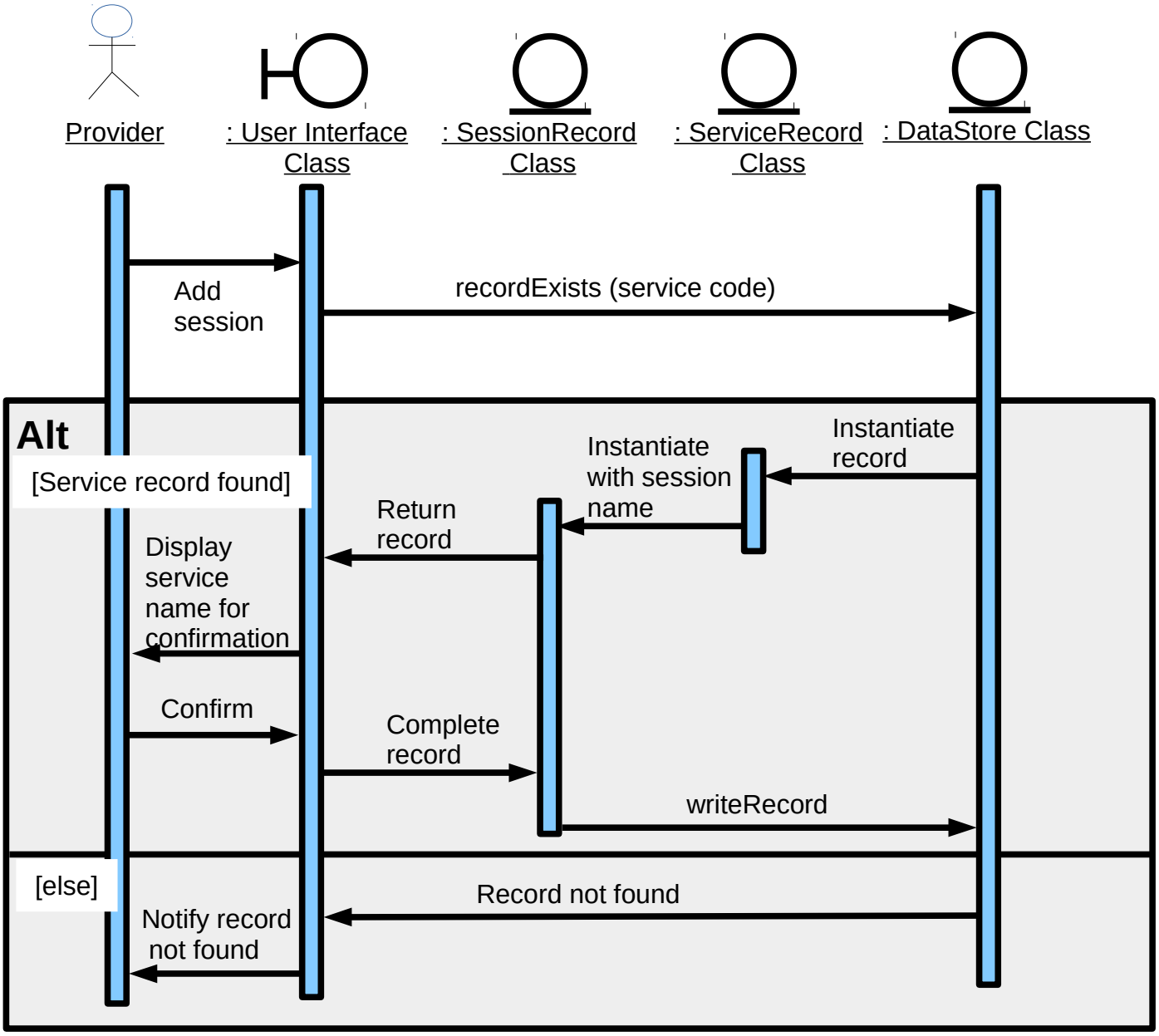
The modify record use case:



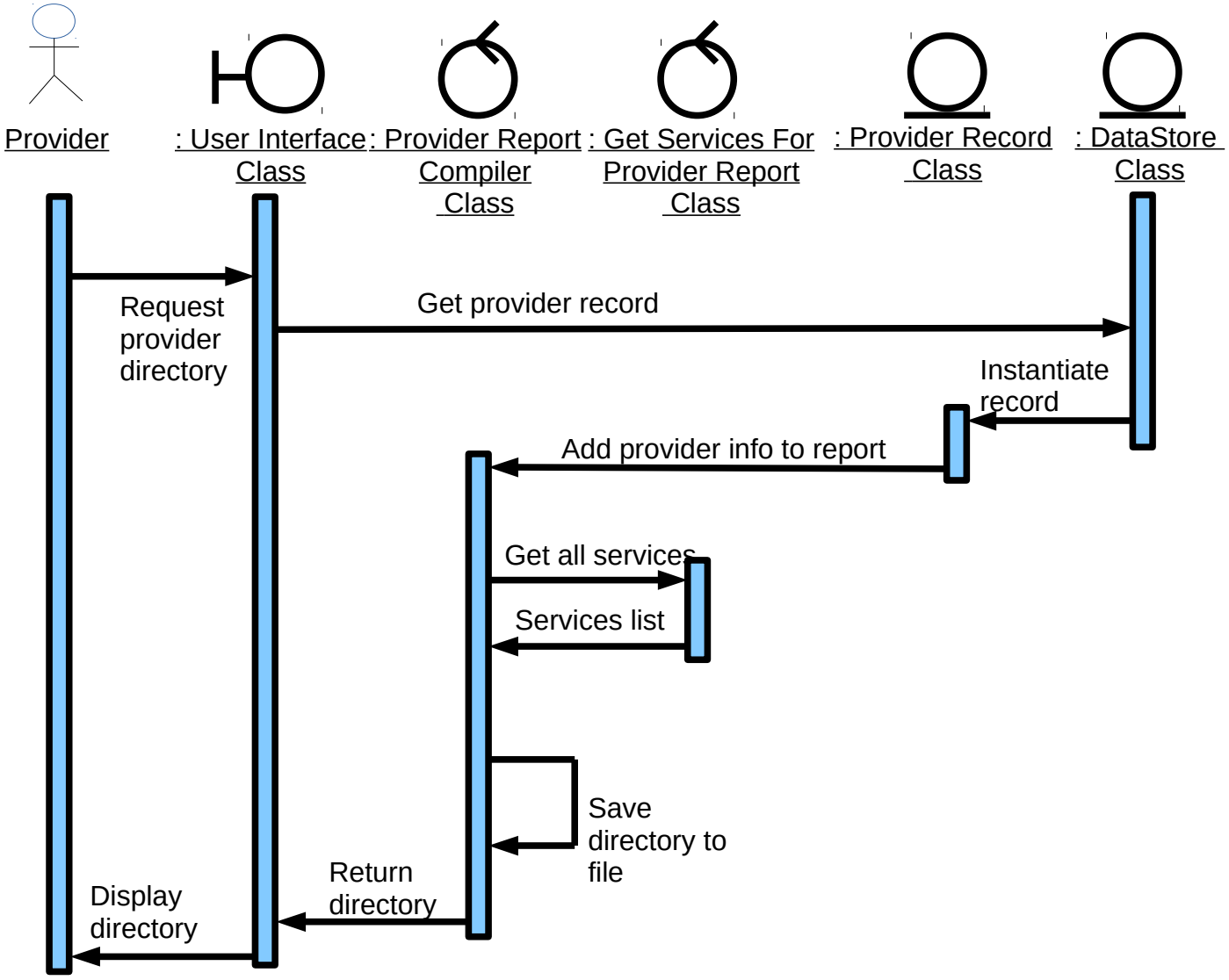
The validate member use case. Note that I have added a recordExists method to the data store. Why use getRecord, which would instantiate the record, when we just want to know whether that record exists? Further evidence of the importance of incremental and iterative methods.



The add session use case:

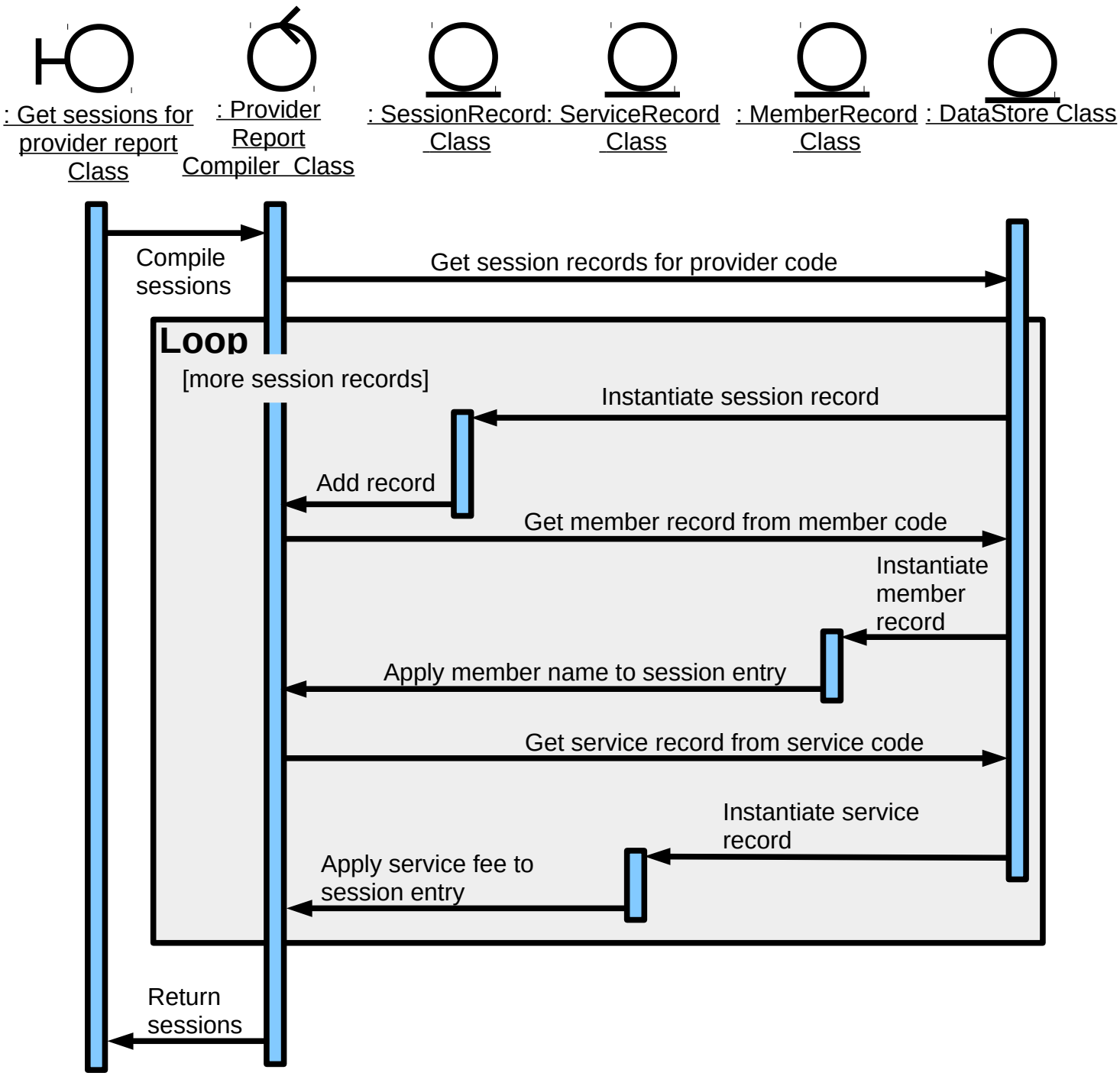


The request provider directory use case requires adding a control class to compile the directory, as this is not a trivial operation, as was the case in the previous use cases:

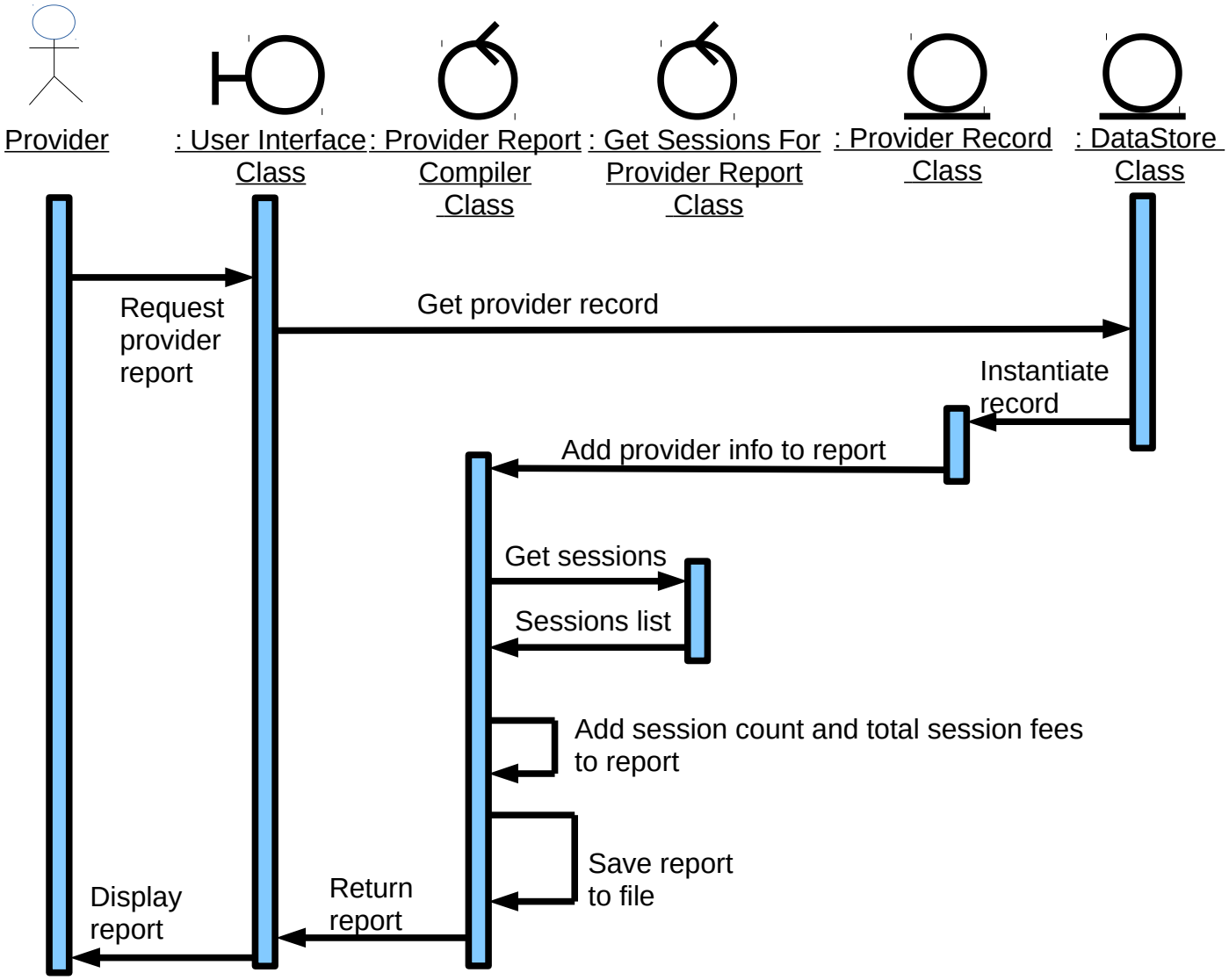


For the request provider report use case, the complexity is high enough that I will define two sequence diagrams: the first of which describes get provider report sessions class:

Get sessions for provider report class:

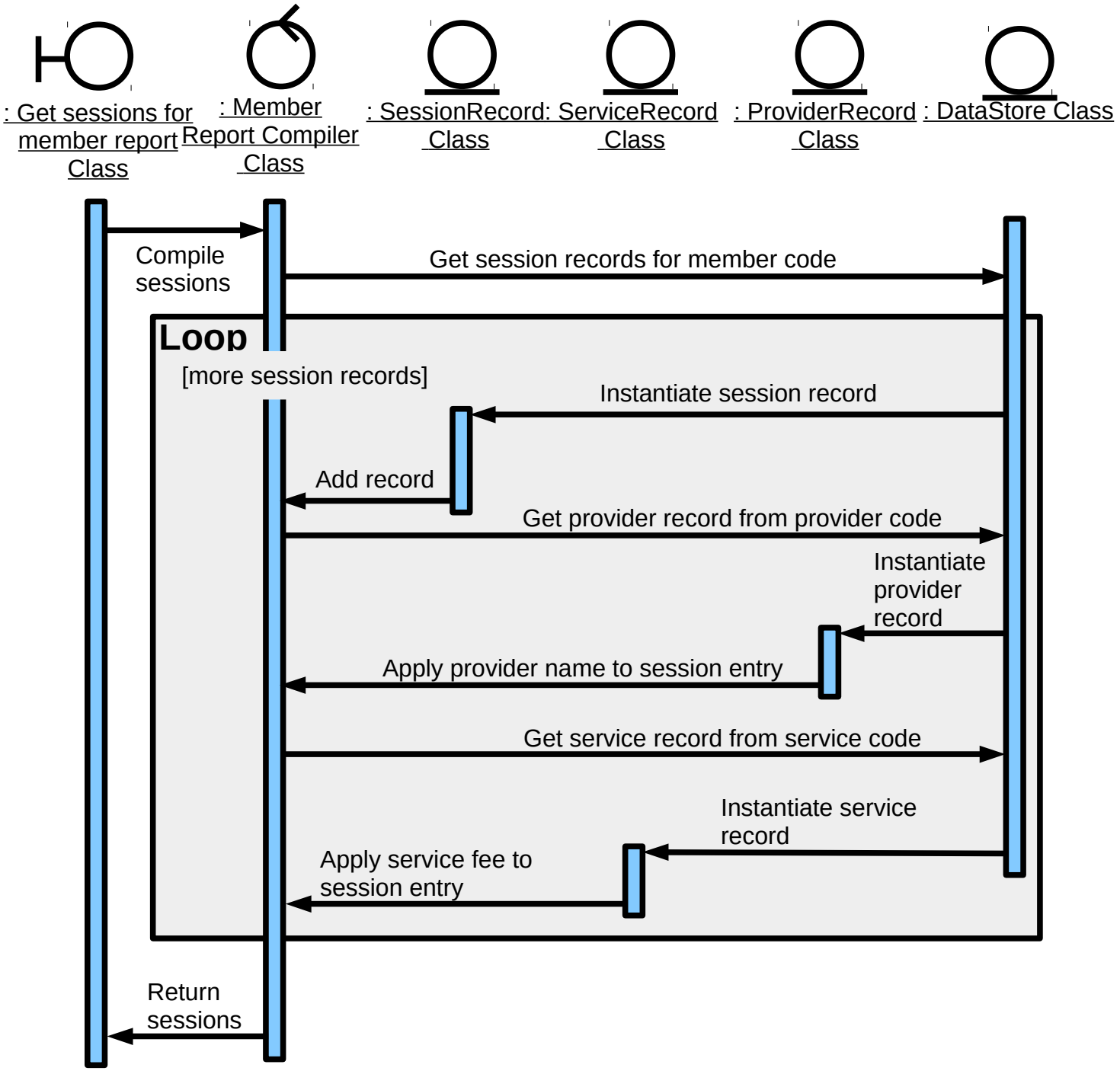


The sequence diagram for the request provider report use case makes use of the previous class:

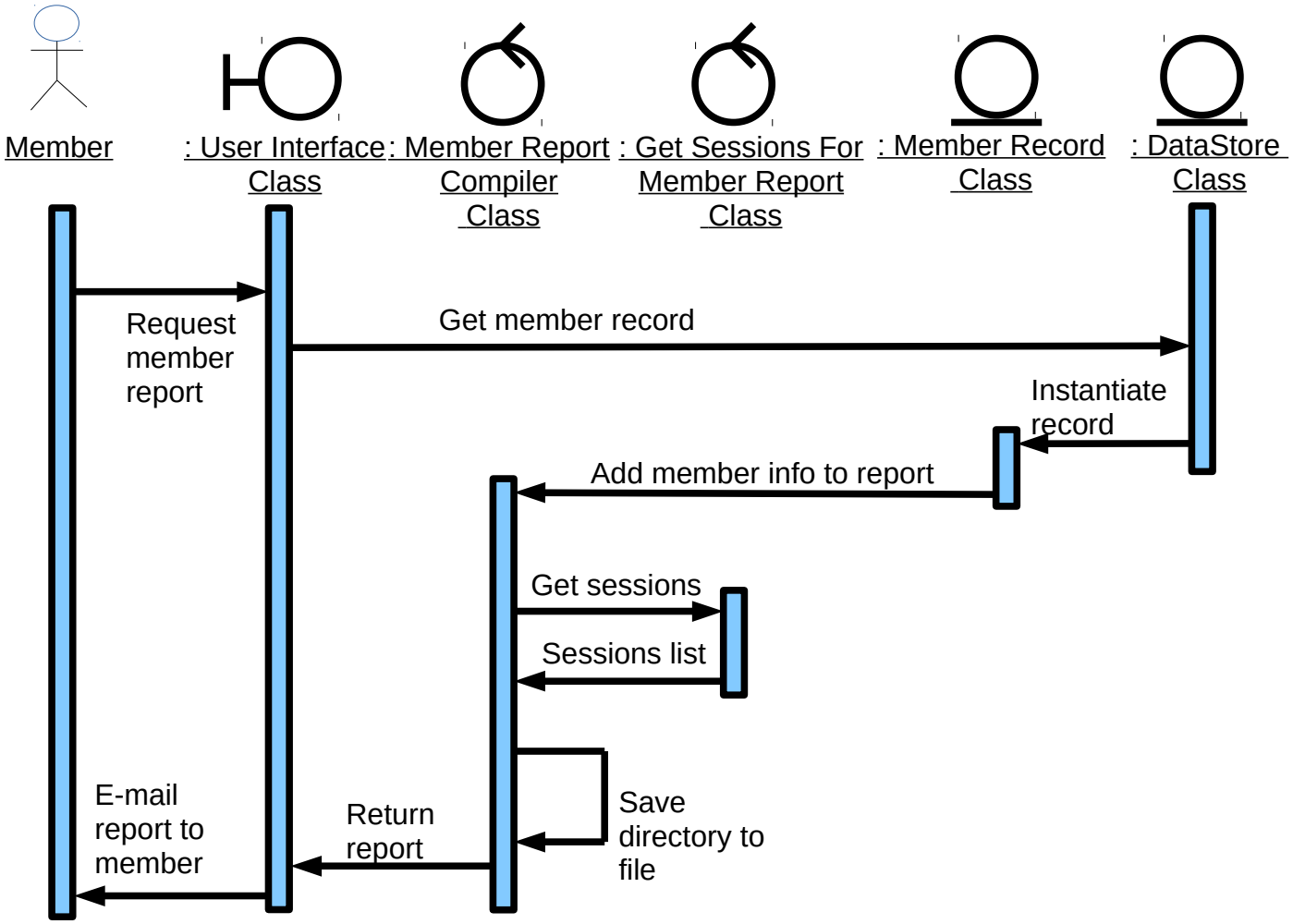


For the member report, we need another separate sequence diagram, for clarity.

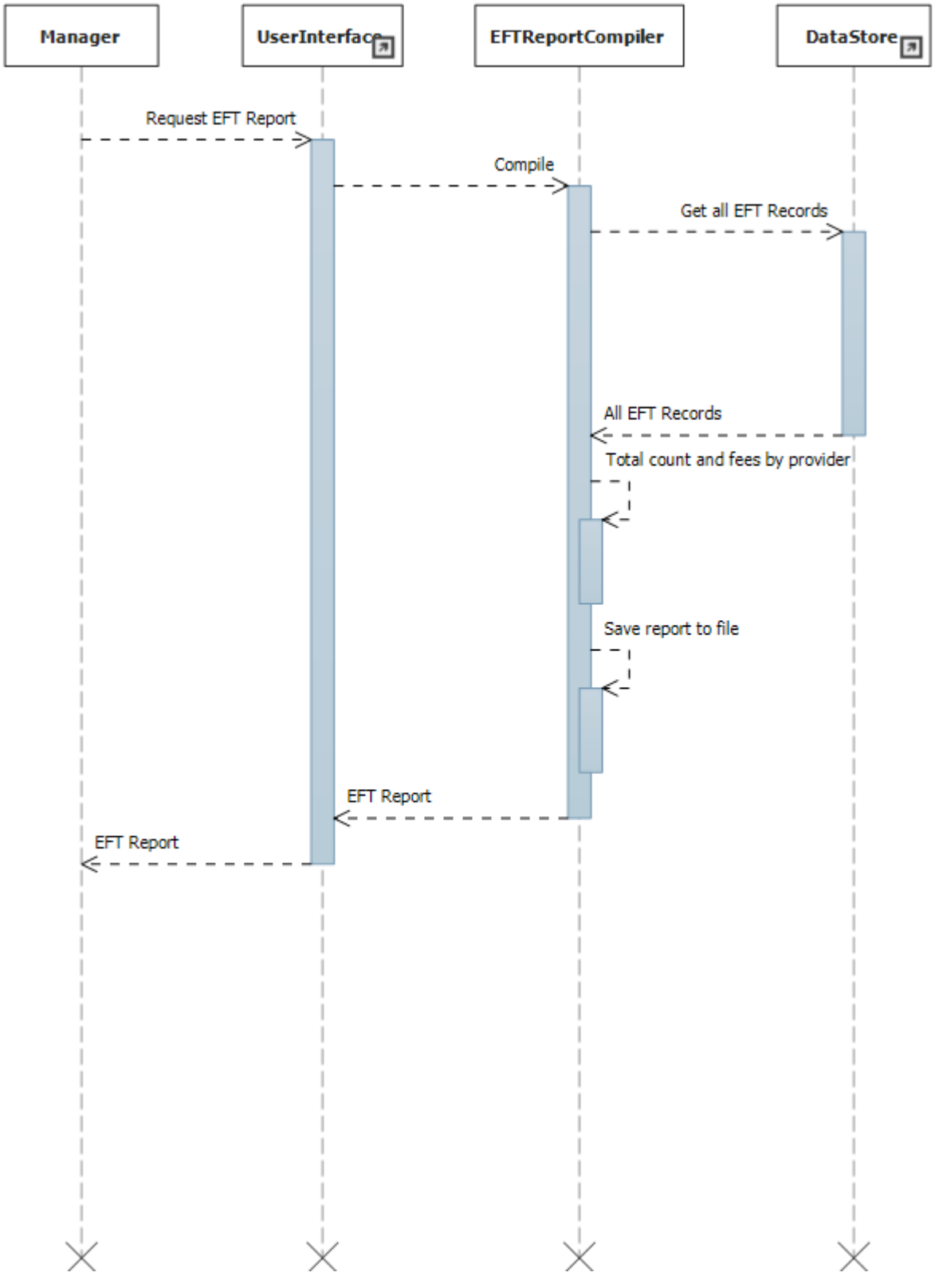
The get sessions for member report class:



The sequence diagram for the request member report use case makes use of the previous class:



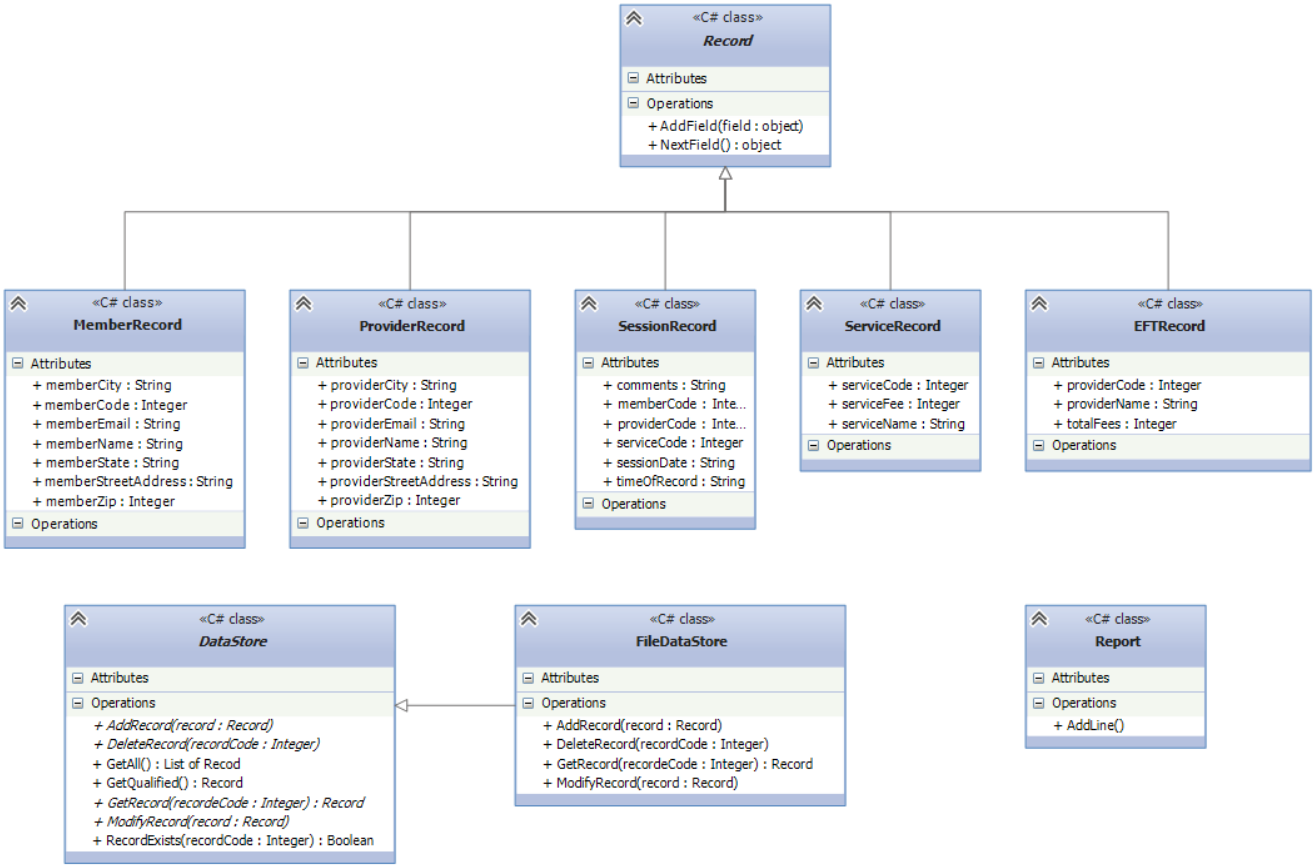
The request accounts payable report use case. Note that this one looks different from the previous ones. This is because I have found that with most tools it is difficult to get diagrams that look like the text asks them to appear. I have found it easier to do by hand. The following was created using Visual Studio's UML tools. It does not let you define line lines as being for boundary, entity, or controller classes, but it was far faster!



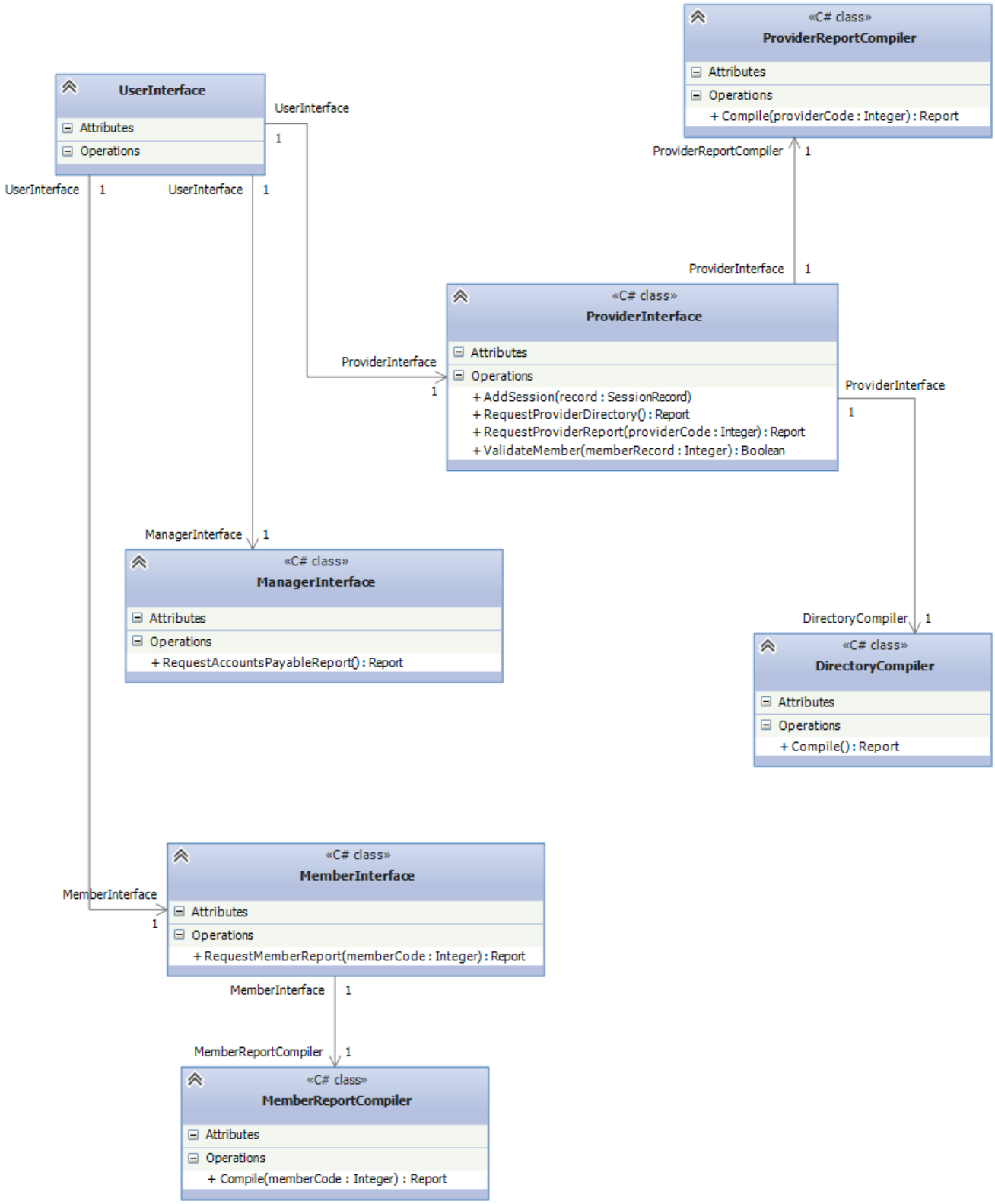
14.16 (Term Project) Starting with your specifications of Problem 12.20 or 13.22, design the Chocoholics Anonymous product (Appendix A). Use the design technique specified by your instructor.

As the first step, I described the classes using Visual Studio's modelling tools. This allows you to generate code stubs for these classes... a big time saver!

Here are all record types that derive from the base record type, a generic report case, and the data store base type with a file data store subclass.



Here are interface/control classes that handle data flow and processing. UserInterface is a stub: I'm unclear as to whether we are designing the user interface, and am assuming it's outside the scope of this assignment. The manager/provider/member interfaces offer service particular to those users. The compiler classes encapsulate the more detailed processing needed to generate reports.



I did further design in a C#-ish pseudocode in visual studio. Here are the data classes:

```
// Represents an abstract base class for all records. This allows implementation of
// a generic data store.
abstract class Record
{
    // Returns the next field as an object if one exists. This is used by the
    // data store to walk through the fields of the record: even though each field
    // is typed as an object, the ToString() method will ensure that it is converted
    // a proper value according to it's actual type...int or string.
    // Returns null if no more fields are found, signalling the end of the record.
       public object NextField()
       {
        // There are a few ways to test for more fields: store them in a list or dictionary,
        // or use reflection to inspect fields of this class.
              if (/*there are more fields */)
            // return next field
         else
             return null

       }

    // Add an object representing the field to the record
    public void AddField(object field)
    {
        // Add record to field, however that is implemented.
    }

}


// Storage format for EFT data.
// Property getters simply return the value.
// Setters apply the value to the field after sanity checking.

public class EFTRecord : Record
{
       public string providerName
       {
              get;
              set;
       }

       public int providerCode
       {
              get;
              set;
       }

    // The total sum of funds to transfer to the provider
       public int transferSum
       {
              get;
              set;
       }

}
```

```
// Storage format for member data.
// Property getters simply return the value.
// Setters apply the value to the field after sanity checking.
class MemberRecord : Record
{
	public int memberCode
	{
		get;
		set
		{
		// Check that the string representation of the member code
		// does not exceed 9 digits.
		}
	}

	public string memberName
	{
		get;
		set
		{
		// Check that the name does not exceed
		// 25 characters.
		}
	}

	public string memberCity
	{
		get;
		set
		{
		// Check that the name does not exceed 14 characters
		}
	}

	public string memberStreetAddress
	{
		get;
		set
		{
		// Make sure that the street address does not exceed
		// 25 characters.
		}
	}

	public string memberState
	{
		get;
		set
		{
		// Make sure that the state does not exceed 2 characters.
		}
	}

	public int memberZip
	{
		get;
		set
		{
```

```
                // Make sure that the string representation of the zip code
                // does not exceed 5 characters.
                }
        }

        public string memberEmail
        {
                get;
                set
                {
                // Make sure that the e-mail address does not exceed
                // 25 characters.
                }
        }

}



/* Storage format for a provider record */
class ProviderRecord : Record
{
    public int providerCode
    {
        get;
        set
        {
            // Check that the string representation of the provider code
            // does not exceed 9 digits.
        }
    }

    public string providerName
    {
        get;
        set
        {
            // Check that the name does not exceed
            // 25 characters.
        }
    }

    public string providerCity
    {
        get;
        set
        {
            // Check that the name does not exceed 14 characters
        }
    }

    public string providerStreetAddress
    {
        get;
        set
        {
            // Make sure that the street address does not exceed
            // 25 characters.
        }
    }
```

```
    public string providerState
    {
        get;
        set
        {
            // Make sure that the state does not exceed 2 characters.
        }
    }

    public int providerZip
    {
        get;
        set
        {
            // Make sure that the string representation of the zip code
            // does not exceed 5 characters.
        }
    }

    public string providerEmail
    {
        get;
        set
        {
            // Make sure that the e-mail address does not exceed
            // 25 characters.
        }
    }

}


// The storage format for service records.

class ServiceRecord : Record
{
       public int serviceCode
       {
              get;
              set
              {
              // Make sure that the string representation of the service code
              // does not exceed 6 characters.
              }
       }

       public string serviceName
       {
              get;
              set
              {
              // Make sure that the service name does not exceed 25 characters.
              }
       }

       public int serviceFee
       {
              get;
              set
```

```
                {
                // Make sure that the string representation of the service fee does not exceed
                // 5 characters.
                }
        }

}

// Storage format for session records.
class SessionRecord : Record
{
        public virtual string timeOfRecord
        {
                get;
                set
                {
                // Make sure that the time of record is a 19 character string in the
                // format MM-DD-YYYY HH:MM:SS
                }
        }

        public virtual string sessionDate
        {
                get;
                set
                {
                // Make sure that the date is a 10 character string in the format
                // MM-DD-YYYY
                }
        }

        public virtual int providerCode
        {
                get;
                set
                {
                // Make sure that the string representation of the provider code does not
                // exceed 9 characters.
                }
        }

        public virtual int memberCode
        {
                get;
                set
                {
                // Make sure that the string representation of the member code does not
                // exceed 9 characters.
                }
        }

        public virtual int serviceCode
        {
                get;
                set
                {
                // Make sure that the string representation of the service code does not
                // exceed 6 characters.
                }
        }
```

```
        public virtual string comments
        {
                get;
                set
                {
                // Make sure that the length of the comments does not exceed 100 characters.
                }
        }

}
```

Now the abstract data store class and it's file subtype:

```
// Abstract, generic, base class for data stores.
// Can perform operations for any format of record, as long
// as they derive from Record.
abstract class DataStore
{
    // Add the record to the data store.
        public abstract void AddRecord(Record record)

    // Retrieve the record matching the code from the data store.
        public abstract Record GetRecord(int recordeCode)

    // Delete the record matching the code from the data store.
        public abstract void DeleteRecord(int recordCode)

    // Overwrite the record in the data store.
        public abstract void ModifyRecord(Record record)

    // Check if a record with the given code exists. Return true
    // if it does, return false if it does not.
        public bool RecordExists(int recordCode)
        {
                if(GetRecord(recordCode) == null)
             return false
         else
             return true
        }

    public (List of Records) GetQualified(/* qualifiers here */)
    {
        /* Did not realise the need for this until the design phase: more
         * evidence of increment/iteration in action. We need a method of
         * getting records by some other criterion besides it's code. Perhaps
         * we want to get a list of all services with a given provider code?
         * This is one area where it would be nice to use a database rather
         * than files. */
    }

    public (List of Records) GetAll()
    {
        /* Another missed method: get all records in the datastore. We need
         * this one to get all services when building the provider directory. */
    }
}
```

```
// File system implementation of DataStore. I'd prefer to use something
// less terrible like a database, but it's what the scenario calls for :)
class FileDataStore : DataStore
{

        public override void AddRecord(Record record)
        {
         // Check that the file does not already exist... it's a fault if it does!
               // Create the file... perhaps the records code is the filename?

         // Assign a unique code to the record

         // Keep looping until end of record
         while(true)
         {
             object field = record.NextField()
             if(field == null) return
             // write field.ToString() to file
         }

         // Close the file
        }

        public override Record GetRecord(int recordeCode)
        {
         // Check that the file exists: it's a fault if it does not!
         // Open the file... perhaps the records code is the filename?

         // Create new record instance
         Record record = new Record()
               while(/* more lines to read */)
              record.AddField(/* input data */)

         // Close the file

         return record
        }

        public override void DeleteRecord(int recordCode)
        {
         // Check that the file exists: it's a fault if it does not!

         // DO NOT DELETE THE FILE: put into a repository

         // Close the file
        }

        public override void ModifyRecord(Record record)
        {
         // Check that the file exists: it's a fault if it does not!

         // DO NOT DELETE THE FILE: put into a repository

         AddRecord(record)
        }

}
```

And the interface designs, excepting the user interface stub:

```
// Provides the interface through which all manager services
// pass

// This interface should include methods to add and delete members,
// providers, services, and sessions, but they'd be simple pass-through
// methods, so for brevity I won't show them here, except for one example
class ManagerInterface
{
       public Report RequestAccountsPayableReport()
       {
              // Compile an EFT report
        Report eftReport = EFTReportCompiler.Compile()

        // Write the eftReport to a file

        // Return for display
        return eftReport
       }

    // Example pass-through method. Pretend there's others for getting, deletion and
modification.
    public void AddMember(MemberRecord memberRecord)
    {
        memberDataStore.AddRecord(memberRecord)
    }
}


// Provides the interfaces through which all member services pass
class MemberInterface
{
    // Return a member report for the member corresponding to the given
    // member code.
    // Requires access to an instance of MemberReportCompiler.
       public Report RequestMemberReport(int memberCode)
       {
              Report report = memberReportCompiler.Compile(memberCode)

        // Write the report to a file, with a file name consisting of the member name
        // followed by the date of the report.

        // Return report for display
        return report
       }

}
```

```
// Provides the interface through which provider services pass.
class ProviderInterface
{

    // Adds the given session to the session data store.
    // This method requires access to the session data store.
       public void AddSession(SessionRecord record)
       {
              sessionDataStore.AddRecord(record)

        // Check for error states
       }

    // Returns true if a member with this code exists. Otherwise,
    // returns false.
    // Requires access to the member data store.
       public bool ValidateMember(int memberRecord)
       {
              return memberDataStore.RecordExists(memberRecord)
       }

    // Returns a provider directory: all available services,
    // their codes, and fees
    // Requires access to a DirectoryCompiler instance
       public Report RequestProviderDirectory()
       {
              Report report = directoryCompiler.Compile()

        // Write the report to a file

        // Return report for display
        return report
       }

    // Returns a provider report for the provider matching
    // the given provider code.
    // Requires access to ProviderReportCompiler and EFTRecord data store instances.
       public Report RequestProviderReport(int providerCode)
       {
              Report report = providerReportCompiler.Compile(providerCode)

        // Write the provider report to a file with a file name
        // starting with the provider name, followed by the
        // date of the report.


        // Write the EFT record for this report
        EFTRecord record
        record.providerCode = // provider code from report
        record.providerName = // provider name from report
        record.transferSum = // total fees from report
        eftDataStore.AddRecord(record)

        // Return report for display
        return report
       }

}
```

A basic stub class used for compiling reports:

```
// A stub representing reports. Reports are an implementation detail that
// I'm skipping here, since we're just designing.
class Report
{
    // Adds the given string to the report as a new line.
       public void AddLine(string line)
       {
              // Insert implementation that addes line to a report here.
       }

    // Insert the given report into this report
    public void AddReport(Report report)
    {
        // Append lines of given report to this report.
    }
}
```

And finally, the report compilers:

```
// Compiles and returns a provider directory: the list of all services,
// their codes, and their fees.
class DirectoryCompiler
{
    // Compile and return a provider directory.
    // Report is a stub class to which we'll just add lines.
    // It's an implementation detail anyways.
    // This method requires access to the provider and service
    // record DataStore instances.
       public Report Compile()
       {
        Report report = new Report()

        // Write some sort of header to the report
        report.AddLine(/* header */)

        // Get list of all service records from the service data store.
        (List of ServiceRecord) services = serviceDataStore.GetAll()

        foreach(ServiceRecord service in services)
        {
            report.AddLine(service.serviceName)
            report.AddLine(service.serviceCode)
            report.AddLine(service.serviceFee)
        }

        return report
       }

}
```

```
// Compiles an electronic funds transfer report in the format
/* described in the appendix:
 *
 * For each provider for which there are EFT records:
 * provider name
 * total number of sessions
 * total fees for those sessions
 *
 * total number of unique providers with EFT records
 * total number of session for all these providers
 * total fees for all these providers
 */
class EFTReportCompiler
{
    /* Compile and return an EFT report */
    public Report Compile()
       {
              // Create the report
        Report report = new Report()

        // Add some sort of header
        report.AddLine(/*header*/)

        // Get all ETF records
        (List of EFTRecord) eftrecords = eftDataStore.GetAll()

        // Get list of unique provider names in this eft records
        uniqueProviderNames = GetUniqueProviderNames(eftrecords)

        // Store the total fees for all providers here
        int totalFeesAll = 0

        // Iterate through the unique provider names
        foreach(string providerName in uniqueProviderNames)
        {
            // Total fees for this provider
            int totalFees = 0;
            // Total consultations for this provider
            int totalConsultations = 0;
            // For each of these provider names, iterate through the eft records
            // to find records for that provider
            foreach(EFTRecord eftrecord in eftrecords)
            {
                // Check for matching provider name
                if(providerName == eftrecord.providerName)
                {
                    totalFees += eftrecord.transferSum
                    totalConsultations++
                }
            }

            // Update the total fees for all providers
            totalFeesAll += totalFees

            // Output the data for this provider to the report
            report.AddLine(providerName)
            report.AddLine(totalConsultations)
            report.AddLine(totalFees)
        }
```

```
        // Output the total number of unique providers to the report
        report.AddLine(uniqueProviderNames.Count)

        // Output the total number of consultaions
        report.AddLine(eftrecords.Count)

        // Output the total fees for all
        report.AddLine(totalFeesAll)

        return report
       }

    // Return a list of the unique provider names found in this list of EFT records
    public (List of providerName) GetUniqueProviderNames((List of EFTRecord) eftrecords)
    {
        // Store unique provider records here
        (List of providerName) providers

        // Iterate through the list of eft records
        foreach(EFTRecord eftrecord in eftrecords)
        {
            if(/* provider name matching this records provider name is not in providers list
*/)
            {
                providers.Add(eftrecord.ProviderName)
            }
        }

        return providers
    }
}

/* Compiles and returns a member report. Format as in the appendix:
 *
 * Member name (25 characters).
 * Member number (9 digits).
 * Member street address (25 characters).
 * Member city (14 characters).
 * Member state (2 letters).
 * Member ZIP code (5 digits).
 *
 * For each service provided, the following details are required:
 * Date of service (MM-DD-YYYY).
 * Provider name (25 characters).
 * Service name (20 characters)
*/
class MemberReportCompiler
{
    // Compiles and returns a member report for the member
    // matching the given member code.
       public Report Compile(int memberCode)
       {
              // Create the report
        Report report = new Report()

        // Add some sort of header
        report.AddLine(/*header*/)

        // Add the member's info
        AddMemberInfo(report, memberCode)
```

```
        // Add the member's sessions to the report
        AddSessions(report, memberCode)

        return report
       }

    // Add the member's info to the report.
    void AddMemberInfo(Report report, int memberCode)
    {
        // Get the provider record
        MemberRecord record = memberDataStore.GetRecord(memberCode)

        // Add the provider's info to the report
        report.AddLine(record.memberName)
        report.AddLine(record.memberNumber)
        report.AddLine(record.memberStreetAddress)
        report.AddLine(record.memberCity)
        report.AddLine(record.memberState)
        report.AddLine(record.memberZip)
        report.AddLine(record.memberEmail)
    }

    // Add the list of sessions to the report
    void AddSessions(Report report, int memberCode)
    {
        // Get all sessions fo this member, for this week
        (List of SessionRecord) sessions = sessionDataStore.GetQualified(/* Get by member code
and date range */)

        foreach(SessionRecord session in AddSessions)
        {
            // Add session date to report
            report.AddLine(session.sessionDate)

            // Get the provider's record, we need it's name
            ProviderRecord providerRecord = providerDataStore.GetRecord(session.providerCode)
            // Add provider's name to report
            report.AddLine(providerRecord.providerName)

            // Get the service record: we need it's name as well
            ServiceRecord serviceRecord = serviceDataStore.GetRecord(serviceRecord.serviceCode)
            // Add service's name to report
            report.AddLine(serviceRecord.serviceName)
        }
    }
}
```

```
/* Compiles a provider report for the provider with the given provider code. */
// This method requires access to all datastore instances.
/*
 * From the appendix, the format is:
 *
 * Provider name (25 characters).
 * Provider number (9 digits).
 * Provider street address (25 characters).
 * Provider city (14 characters).
 * Provider state (2 letters).
 * Provider ZIP code (5 digits).
 *
 * For each service provided, the following details are required:
 * Date of service (MM-DD-YYYY).
 * Date and time data were received by the computer (MM-DD-YYYY HH:MM:SS).
 * Member name (25 characters).
 * Member number (9 digits).
 * Service code (6 digits).
 * Fee to be paid (up to $999.99).
 *
 * Total number of consultations with members (3 digits).
 * Total fee for week (up to $99,999.99). */
class ProviderReportCompiler
{
    /* Compiles and returns a provider report for the provider
     * with the given provider code. */
       public Report Compile(int providerCode)
       {
        // Create the report
              Report report = new Report()

        // Add some sort of header
        report.AddLine(/* header */)

        // Add provider's information
        AddProviderInfo(report, providerCode)

        // Add all sessions and their totals.
        AddSessions(report, providerCode)

        return report
       }

    // Add the provider's info to the report.
    void AddProviderInfo(Report report, int providerCode)
    {
        // Get the provider record
        ProviderRecord record = providerDataStore.GetRecord(providerCode)

        // Add the provider's info to the report
        report.AddLine(record.providerName)
        report.AddLine(record.providerNumber)
        report.AddLine(record.providerStreetAddress)
        report.AddLine(record.providerCity)
        report.AddLine(record.providerState)
        report.AddLine(record.providerZip)
        report.AddLine(record.providerEmail)
    }
```

```
    // This is the "Get sessions for provider report class" in the analysis worflow.
    // It assembles and adds the data for a session to the report.
    void AddSessions(Report report, int providerCode)
    {
        // Get all sessions by this provider, for this week
        (List of SessionRecord) sessions = sessionDataStore.GetQualified(/* Get by provider
code and date range */)

        // Total the number of sessions and their fees
        int totalSessions = 0;
        int totalFees = 0;

        foreach(SessionRecord session in sessions)
        {
            // Add session dates to report
            report.AddLine(session.sessionDate)
            report.AddLine(session.timeOfRecord)

            // Add member's name and code to report.
            MemberRecord member = memberDataStore.GetRecord(session.memberCode)
            report.AddLine(member.memberName)
            report.AddLine(member.memberCode)

            // Add service's code to report
            report.AddLine(session.serviceCode)

            // Get the service record for it's fee
            ServiceRecord service = serviceDataStore.GetRecord(session.serviceCode)
            report.AddLine(service.serviceFee)

            // Increment the session count
            totalSessions++

            // Add the fee of the session to the total
            totalFees += service.serviceFee
        }

        // Add the total number of sessions and total fees to the report
        report.AddLine(totalSessions)
        report.AddLine(totalFees)
    }
}
```

15.33 (Term Project) Draw up black-box test cases for the product you specified in Problem 12.20 or 13.22. For each test case, state what is being tested and the expected outcome of that test case.

I will discuss testing of the specification document (Problem 12.20). This requires a team of inspectors to review the document to ensure that it meets requirements of the product. The team must record, but not fix, each fault: fixing them at the same time will make the inspection process too long.

The first, simplest thing to test that all data values listed in the dictionary have their data formats and maximum lengths specified. For example, are provider codes 9 digit numbers? Are service codes 9 digit numbers? Are session dates 10 characters and in the format DD-MM-YYYY? The data formats and sizes in the dictionary all appear to be correct, but a team of inspectors may see something I missed.

The second thing to test is that invalid data is handled correctly. Does provider invalid provider, member, and service codes result in an error message, and not failure? This seems to be the case, but a team may see something I missed as well.

The third thing to inspect is that the acceptance requirements are met.

Acceptance Requirement	Requirement met?
“The data from a provider’s terminal must be simulated by keyboard input and data to be transmitted to a provider’s terminal display must appear on the screen. A manager’s terminal must be simulated by the same keyboard and screen”	Yes, if you consider that much of these are low level details to be considered in the design and implementation workflows. For example, users can request reports and records, add sessions, validate members, and reports and records are returned to the user. In this workflow the actual implementation of keyboard input and terminal output is omitted.
“Each member report must be written to its own file; the name of the file should begin with the member name, followed by the date of the report”	Yes
“Each provider report must be written to its own file; the name of the file should begin with the provider name, followed by the date of the report”	Yes
“None of the files should actually be sent as e-mail attachments”	Yes – no e-mail messages or attachments of any kind are sent.
“As for the EFT data, all that is required is that a file be set up containing the provider name, provider number, and the amount to be transferred”	Yes

16.15 (Term Project) Suppose that the product for Chocoholics Anonymous in Appendix A has been implemented exactly as described. Now the product has to be modified to include endocrinologists as providers. In what ways will the existing product have to be changed? Would it be better to discard everything and start again from scratch? Compare your answer to the answer you gave to Problem 1.19.

My answer to the original question is unchanged: there would be no need to discard anything. Adding endocrinologists would require adding a new provider record with a unique code for each endocrinologist. Then each service offered by the endocrinologist would need a new service record with a new unique service code, the name of the service, and it's fee.

Once this is done, the provider and service are treated the same as any other by the system.