
S60 Platform: Landmarks Search API Specification

Version 1.0
July 12, 2005

S60 platform

Legal notice

Copyright © 2005 Nokia Corporation. All rights reserved.

Nokia and Forum Nokia are registered trademarks of Nokia Corporation. Other product and company names mentioned herein may be trademarks or trade names of their respective owners.

Disclaimer

The information in this document is provided “as is,” with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Furthermore, information provided in this document is preliminary, and may be changed substantially prior to final release. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

License

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.

Contents

1.	Purpose	6
1.1	List of the required interfaces	6
1.2	Constraints	6
2.	Technical specification	7
2.1	Type of the interface	7
2.2	Interface class structure	7
2.2.1	Foundation classes	7
2.2.2	Search criterion classes	8
2.2.3	Displayable data classes	10
2.2.4	Multiple database search classes	11
2.3	Usage	12
2.3.1	Protocol	12
2.3.2	Error handling	18
2.3.3	Memory overhead	18
2.3.4	Extensions to the API	18
2.4	Example	19
2.4.1	Synchronous search using iterator	19
2.4.2	Synchronous search using display data	19
2.4.3	Asynchronous search	20
2.4.4	Using composite criteria	23
2.4.5	Synchronous search in multiple databases	24
2.4.6	Asynchronous search in multiple databases	24
2.5	Detailed description	28
2.5.1	CPosLandmarkSearch	28
2.5.2	CPosLmSearchCriteria	33
2.5.3	CPosLmTextCriteria	34
2.5.4	CPosLmCategoryCriteria	37
2.5.5	CPosLmCatNameCriteria	39
2.5.6	CPosLmAreaCriteria	40
2.5.7	CPosLmNearestCriteria	41
2.5.8	CPosLmCompositeCriteria	43
2.5.9	CPosLmIdListCriteria	46
2.5.10	CPosLmDisplayData	48
2.5.11	CPosLmDisplayItem	50
2.5.12	CPosLmMultiDbSearch	52
2.6	Code architecture	62
3.	Terms and abbreviations	63

4.	References	64
5.	Evaluate this resource	65

Change history

July 12, 2005	Version 1.0	Initial document release Revision on August 29, 2006: minor editorial changes

1. Purpose

The purpose of the Landmarks Search API is to extend the Landmarks API (see reference document [2]) to enable searching for landmarks or landmark categories that match certain criteria. For instance, a client can search for all restaurants in a landmark database.

The Landmarks Search API is used mainly by end-user applications.

The Landmarks Search API is available from S60 3rd Edition onwards.

1.1 List of the required interfaces

The Landmarks Search API is an extension to the Landmarks API (see reference document [2]) and cannot be used alone.

The Landmarks Search API utilizes position classes and generic position field IDs from the Location Acquisition API (see reference document [1]).

The Landmarks API uses ECom (see reference document [3]) to look up and load an implementation, which can search the database specified by a client.

The DBMS API is used to perform searches in local landmark databases.

1.2 Constraints

Landmarks Search API is valid for all platforms running on Symbian OS v9.1 or later.

2. Technical specification

2.1 Type of the interface

The type of this interface is method-call (the interface uses only local objects).

The Landmarks Search API loads the implementation at run time but the implementation consists only of local objects.

2.2 Interface class structure

The following sections describe the Landmarks Search API class structure. UML diagrams are used to present the classes and their dependencies.



Note: The UML diagrams do not show all the available functions, and some or all function parameters may be left out.

2.2.1 Foundation classes

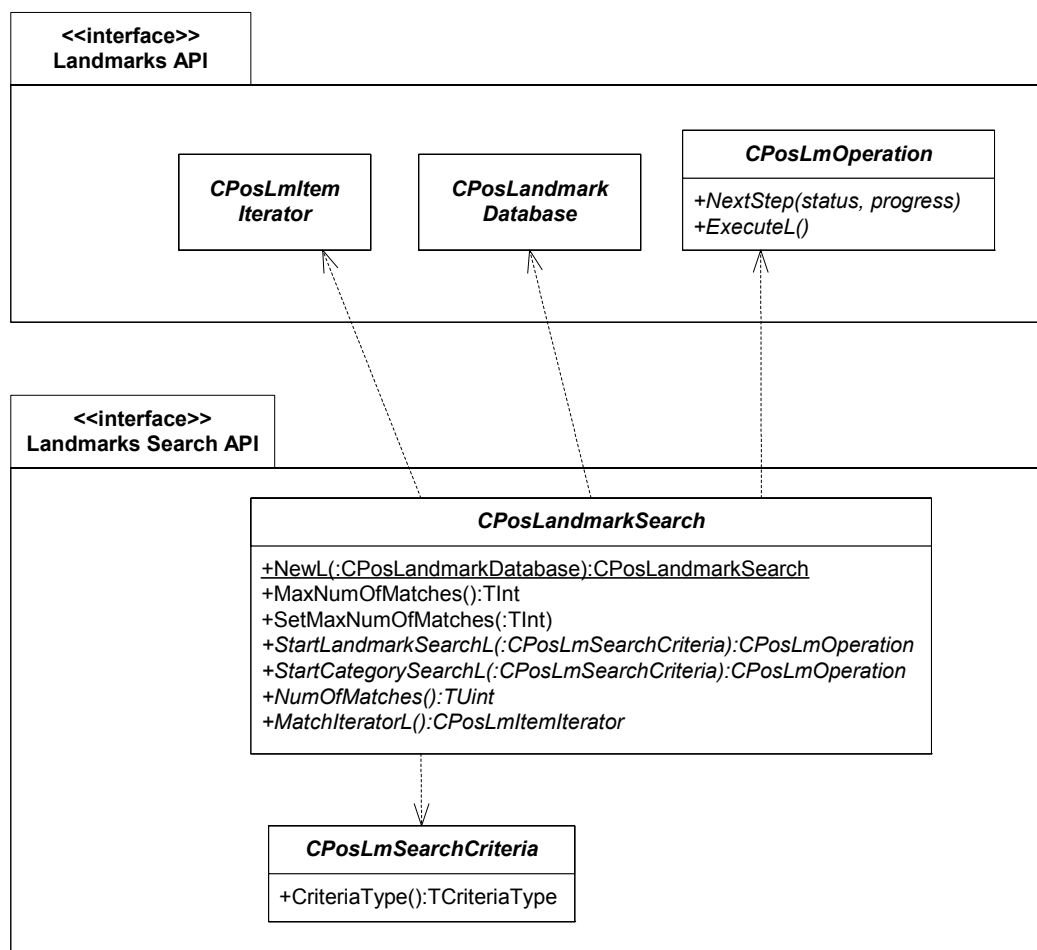


Figure 2.1: Landmarks Search API foundation classes

The foundation of the Landmarks Search API consists of two classes: `CPosLandmarkSearch` and `CPosLmSearchCriteria`.

`CPosLandmarkSearch` is mainly used to start searches and to obtain the search matches. The client has to specify the landmark database to search when `CPosLandmarkSearch` is instantiated.

`CPosLandmarkSearch` returns a `CPosLmOperation` instance, which is used to execute the search operation incrementally or all at once. If the operation is run incrementally, the client can check the operation progress between the incremental steps.

`CPosLmSearchCriteria` is a base class for search criterion classes in the Landmarks Search API. Criterion classes are used to specify what criteria a landmark must fulfill in order to be a search match. There are several search criterion classes in the Landmarks Search API.

`CPosLandmarkDatabase`, `CPosLmItemIterator`, and `CPosLmOperation` are parts of the Landmarks API and they are described further in reference document [2].

2.2.2 Search criterion classes

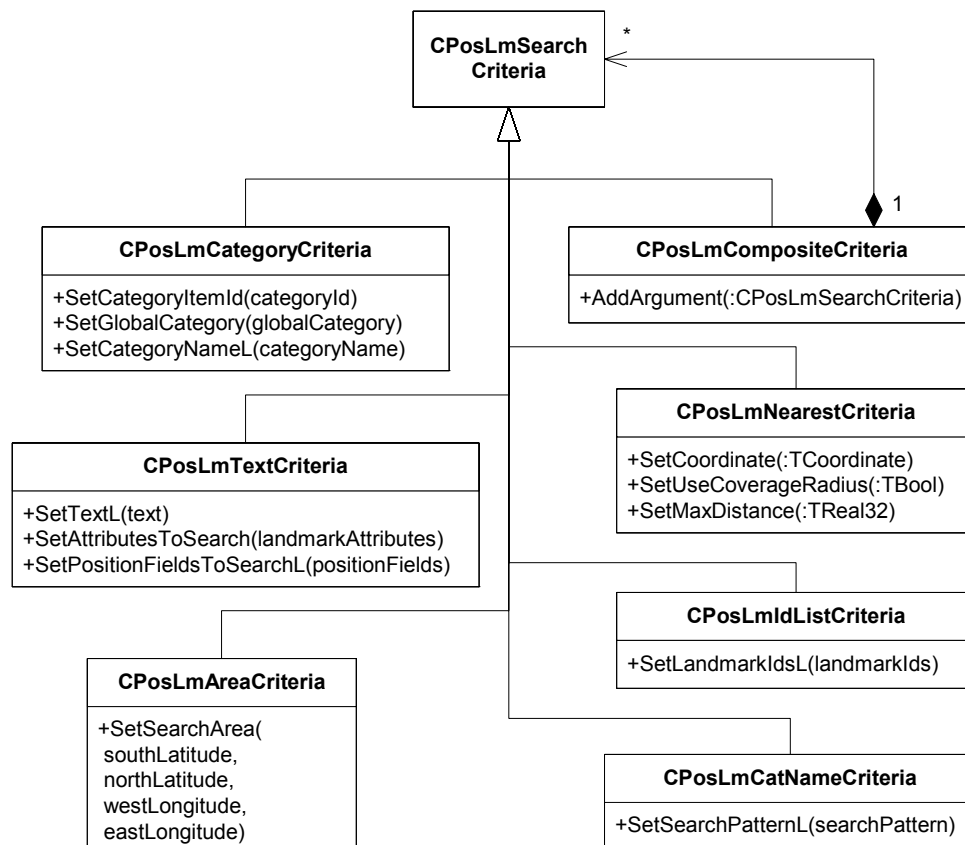


Figure 2.2: Landmark search criterion classes

Criterion class	Description
CPosLmCategoryCriteria	Used to search for landmarks which contain a certain category, or landmarks which do not contain any categories.
CPosLmTextCriteria	Used to search for landmarks which contain a certain text.
CPosLmAreaCriteria	Used to search for landmarks which reside in a certain area.
CPosLmNearestCriteria	Used to find the landmarks which are closest to a certain coordinate.
CPosLmCompositeCriteria	Used to search for landmarks by combining multiple search criteria.
CPosLmIdListCriteria	Used if the client only wants to search a subset of the landmarks in a database.
CPosLmCatNameCriteria	Used to search for landmark categories with a certain name.

2.2.3 Displayable data classes

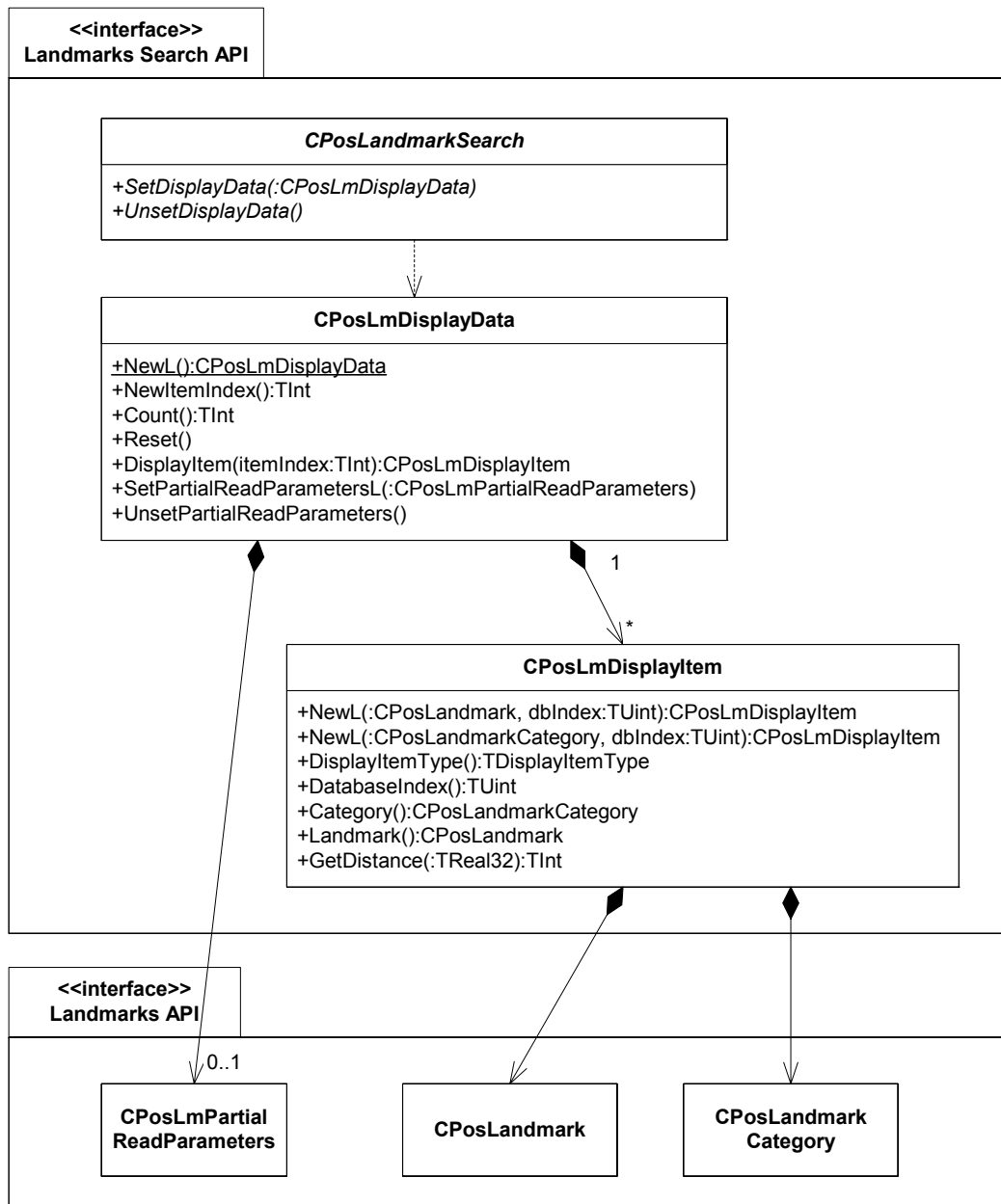


Figure 2.3: Landmark search displayable data classes

When using the landmark item iterator to retrieve search matches, the matches will not be sorted until the search operation has completed. If the client needs a sorted list of the matches before the search has completed, for example, it wants to show matches in a sorted list as they are found to the application user, the display data feature should be used instead.

Before a search is started, the client sets `CPosLmDisplayData` to be used in the search. When the search is executing, the matching categories or landmarks are read from the database and added to `CPosLmDisplayData`. `CPosLmDisplayItem` holds the matching `CPosLandmark` or `CPosLandmarkCategory` together with the index of the database where the

match was found. The database index is only used when searching in several databases.

2.2.4 Multiple database search classes

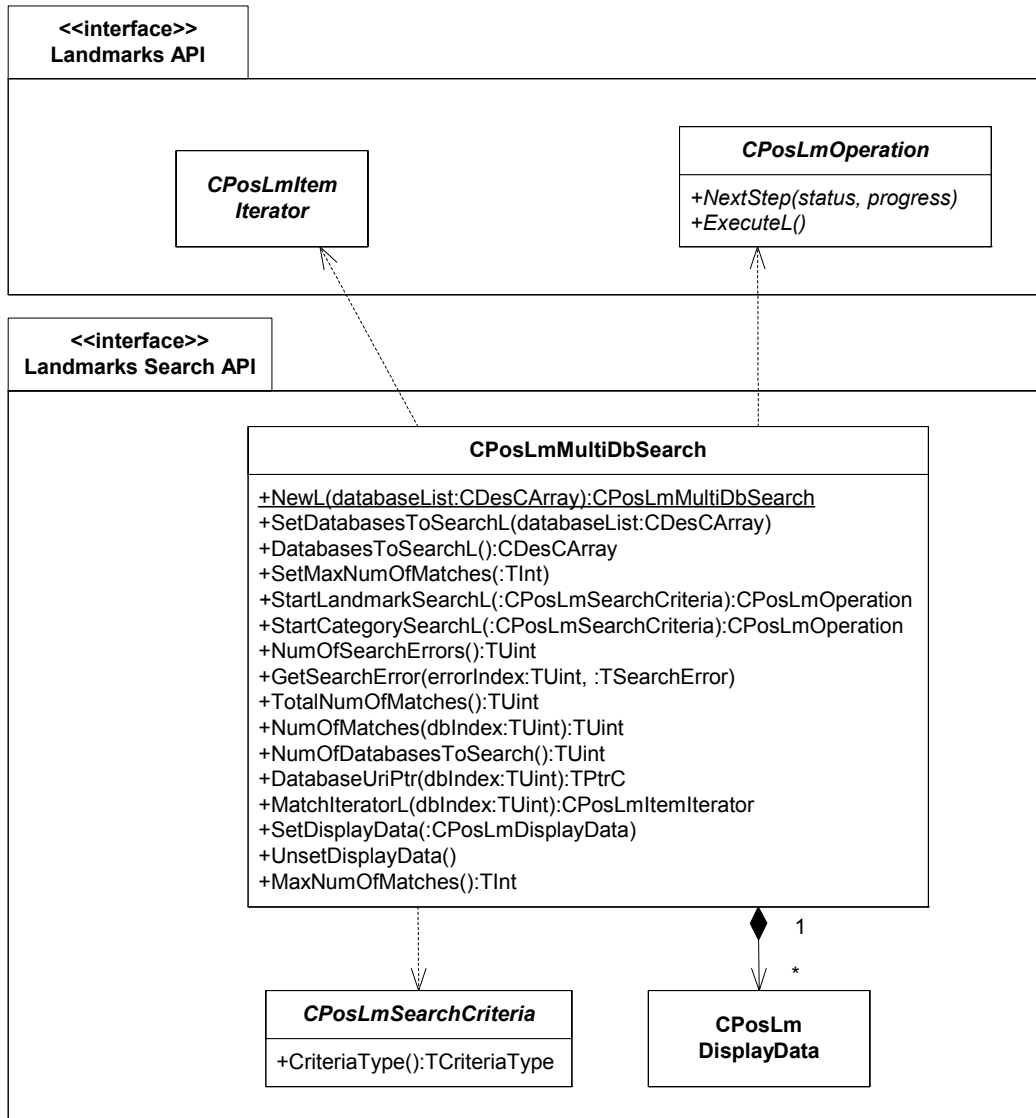


Figure 2.4: Landmark multiple database search classes

`CPosLmMultiDbSearch` is used to search for landmarks and landmark categories in several landmark databases. The client specifies the URIs of the landmark databases to be searched.

`CPosLmMultiDbSearch` returns a `CPosLmOperation` instance, which is used to execute the search operation incrementally or all at once. If the operation is run incrementally, the client can check the operation progress between the incremental steps.

`CPosLmItemIterator` with the IDs of the search matches can be obtained from `CPosLmMultiDbSearch`. It is also possible for the client to set `CPosLmDisplayData` to `CPosLmMultiDbSearch` before the search is started. `CPosLmDisplayData` will be filled with one `CPosLmDisplayItem` for each match, see Section 2.2.3.

2.3 Usage

2.3.1 Protocol

To create a `CPosLandmarkSearch` instance, the client must have a handle to an open database, that is, a `CPosLandmarkDatabase` instance. This handle is passed to `CPosLandmarkSearch::NewL()`.

It is also possible to search multiple landmark databases, see Section 2.3.1.7.

2.3.1.1 *Configuring a search*

`CPosLandmarkSearch` can be configured by setting certain parameters. The parameters are described in this section.

2.3.1.1.1 Maximum number of matches

The client can set a maximum number of matches by calling `CPosLandmarkSearch::SetMaxNumOfMatches()`. The search operation stops if the maximum number of matches is reached.

If the client does not set a maximum, the search returns all the matches in the database. The client can also set this explicitly by passing `KPosLmMaxNumOfMatchesUnlimited` to `CPosLandmarkSearch::SetMaxNumOfMatches()`.

2.3.1.1.2 Search previous matches only

If the `CPosLandmarkSearch` instance has just finished a search, the client can specify that only these matches should be considered in the next search. This is specified by passing a Boolean to `CPosLandmarkSearch::StartLandmarkSearchL()` or `StartCategorySearchL()` when the search is started. In that way, the client can refine its search, for instance, if there are too many matches.

Searching previous matches is useful if there are too many matches and the application user wants to narrow down the search before listing all matches.

When searching for landmarks, it is also possible to use `CPosLmIdListCriteria` (see Section 2.3.1.2.6) to search previous matches that are not immediate previous matches.

2.3.1.1.3 Specifying sort preference

The client can specify that the matching landmarks or landmark categories should be sorted.

One overload of `CPosLandmarkSearch::StartLandmarkSearchL()` takes `TPosLmSortPref` as parameter. In `TPosLmSortPref`, the client can specify whether the landmarks should be sorted by name in ascending or descending order.

`CPosLandmarkSearch::StartCategorySearchL()` takes `CPosLandmarkCategory::TCategorySortPref` as parameter. `TCategorySortPref` can be set to no sorting, ascending by category name or descending by category name.

2.3.1.1.4 Specifying display data

The client can specify a displayable data collection that will be populated with the matching landmarks or landmark categories during the search. This is specified by creating a `CPosLmDisplayData` instance and passing it to `CPosLandmarkSearch::SetDisplayData()`. The display data object will be reset each time a new search is started.

The client can specify that only partial landmark data will be read from the database by calling `CPosLmDisplayData::SetPartialReadParametersL()`. If the client does not set partial read parameters, full landmark data will be read. The client can also set this explicitly by calling `CPosLmDisplayData::UnsetPartialReadParametersL()`.

When searching for categories, full category data is always read from the database.

The client can unset a previously set display data by calling `CPosLandmarkSearch::UnsetDisplayData()`.

2.3.1.2 Specifying search criteria

Criterion classes are used to specify what landmarks to search for. The criteria are passed to `CPosLandmarkSearch::StartLandmarkSearchL()` or `StartCategorySearchL()` when the search is started. In this section, each criterion class is explained.

2.3.1.2.1 CPosLmCategoryCriteria

`CPosLmCategoryCriteria` is used to search for landmarks which belong to a certain category. The category is specified in one of the following ways:

- By category ID: A category in a landmark database is identified by an ID, which is unique within the database.
- By global category: There is a globally defined list of categories, which can be used in multiple databases. The client can specify one of these by global ID.
- By category name: The client can specify the name of the category. The name search is case sensitive. Wild card patterns are not supported.

If no category is specified, the search operation retrieves uncategorized landmarks.

2.3.1.2.2 CPosLmTextCriteria

`CPosLmTextCriteria` is used to search for landmarks which contain a certain text. The criterion is defined by providing a text to search for and the position fields and text attributes to search in each landmark. For example, the user can search for "Chinese" in the landmark description and landmark name attributes.

The search is case insensitive.

If no position fields or landmark attributes to search are specified, all the position fields and landmark attributes are searched in each landmark.

Wild card characters are supported in the search string: "?" matches any single character and "*" matches zero or more consecutive characters.

2.3.1.2.3 CPosLmAreaCriteria

`CPosLmAreaCriteria` is used to search for landmarks which reside in a certain area. The search area is defined by providing two latitude and two longitude values that specify the borders of the area.



Note: This search does not consider landmark coverage radius (coverage radius is defined in reference document [2]).

The area borders must fulfill the following rules:

- $-90 \leq \text{aSouthLatitude} \leq \text{aNorthLatitude} \leq +90$
- $-180 \leq \text{aWestLongitude} < +180$
- $-180 \leq \text{aEastLongitude} \leq +180$

The east border longitude can be less than the west border longitude. This defines an area, which crosses the 180 meridian.

If the east and west border longitudes are equal, only landmarks that lie on the specified longitude are returned. Similarly, if the north and south border latitudes are equal, only landmarks that lie on the specified latitude are returned.

If west longitude is set to -180 and east longitude is set to +180, all longitudes are included in the search.

2.3.1.2.4 CPosLmNearestCriteria

`CPosLmNearestCriteria` is used to find the landmarks which are closest to a certain coordinate. By default, the matches returned in the search are sorted in an ascending distance order.

Since this operation returns all the landmarks in the database by default, it is recommended to combine the nearest criteria with a limit on the number of matches. The match limit is set by calling

```
CPosLandmarkSearch::SetMaxNumOfMatches()
```

It is often a good idea to specify a maximum distance to narrow down the search. This is done by calling `CPosLmNearestCriteria::SetMaxDistance()`.

By default, the coverage radius of the landmarks is not considered in the search; that is, the distance to the landmark center point is used. The client can change this behavior by calling

```
CPosLmNearestCriteria::SetUseCoverageRadius(). If ETTrue is passed to this function, coverage radius is considered; that is, the effective distance is the distance to the landmark center point minus the coverage radius. If the search coordinate lies within a landmark's coverage area, the effective distance is zero.
```

2.3.1.2.5 CPosLmCompositeCriteria

`CPosLmCompositeCriteria` is used to search for landmarks by combining multiple search criteria. For instance, to search for all restaurants in the area, this class can be used to combine `CPosLmAreaCriteria` and `CPosLmCategoryCriteria`.

The client combines the criteria by passing each criterion instance to `CPosLmCompositeCriteria::AddArgument()`.

If `CPosLmNearestCriteria` is used and no sort preference is specified, the result will be sorted by distance. If more than one `CPosLmNearestCriteria` are combined using `CPosLmCompositeCriteria`, the sort order will be undefined unless a sort preference is specified.



Note: It is not allowed to use nested composite criterion.

2.3.1.2.6 CPosLmIdListCriteria

`CPosLmIdListCriteria` is used if the client only wants to search a subset of the landmarks in a database. The client passes a list of landmark IDs to specify which landmarks to include in the search.

This criterion must be combined with other search criteria using `CPosLmCompositeCriteria` (Section 2.3.1.2.5).

For example, if this criterion is combined with `CPosLmTextCriteria`, the search operation searches the landmarks specified in the ID list criterion and returns those that match the given text string.



Note: Only one ID list criterion is allowed in a composite criterion.

2.3.1.2.7 CPosLmCatNameCriteria

`CPosLmCatNameCriteria` is used to search for landmark categories with a certain name. Wild card characters in the search string are supported: "?" matches any single character and "*" matches zero or more consecutive characters.

The search is case insensitive.

2.3.1.3 Starting a search

A search for landmarks is started by calling `CPosLandmarkSearch::StartLandmarkSearchL()`.

A search for landmark categories is started by calling `CPosLandmarkSearch::StartCategorySearchL()`.

Both functions return a `CPosLmOperation` instance, which is used to execute the search operation. The operation can either be executed all at once or in incremental steps. An active object can be used to incrementally run the search in the background.

`CPosLmOperation::ExecuteL()` runs the operation all at once and `CPosLmOperation::NextStep()` runs the operation incrementally.

When the search is complete, the client must delete the `CPosLmOperation` object. Instead of calling `CPosLmOperation::ExecuteL()`, the client can call the utility function `ExecuteAndDeleteLD()` which also deletes the operation object.

For example:

```
ExecuteAndDeleteLD(search->StartLandmarkSearchL(criteria));
```

2.3.1.4 *Checking incremental search progress and status*

If the search is run incrementally, the client is informed of the search progress. The client passes a `TReal32` variable to `CPosLmOperation::NextStep()` which contains the progress when `NextStep()` completes.

Progress is a floating point number in the interval `[0.0, 1.0]`. `0.0` indicates that the operation has not started and `1.0` indicates that the operation has completed.

The client also passes `TRequestStatus` to `NextStep()`. The request status is set to `KPosLmOperationNotComplete` if the step has completed, but more steps are needed before the operation is complete. The request status is `KErrNone` if the operation has finished successfully. The status is set to an error code if the operation has failed.

2.3.1.5 *Retrieving search matches from an iterator*

The IDs of the matches from the search can be retrieved by calling `CPosLandmarkSearch::MatchIteratorL()`. It is possible to call `CPosLandmarkSearch::MatchIteratorL()` also during a search to retrieve any matches encountered so far, but it is not guaranteed that the matches are sorted. However, the matches in the iterator are always sorted, also during a search, if a display data is set to `CPosLandmarkSearch` before the search is started.

2.3.1.6 *Retrieving search matches from display data*

If display data has been set before the search is started, all matches from the search can be retrieved by calling `CPosLmDisplayData::DisplayItem()` for all the indexes in the interval `[0, CPosLmDisplayData::Count() - 1]`. By calling `CPosLmDisplayItem::Landmark()` or `CPosLmDisplayItem::Category()`, the client gets access to the match.

During a search, `CPosLmDisplayData::NewItemIndex()` can be called repeatedly to get the index of each new match found and `CPosLmDisplayData::DisplayItem()` can be called for each new index.

The matches in the display data are always sorted, even during a search.

2.3.1.7 *Searching multiple databases*

Searching for landmarks or landmark categories in multiple landmark databases is rather similar to searching in one database. However, there are some restrictions on the criterion classes and some extra functionality needed to handle several databases.

The client creates a `CPosLmMultiDbSearch` instance by passing an array containing the URIs of the landmark databases to search to `CPosLmMultiDbSearch::NewL()`.

The client can also change which databases to search by calling `CPosLmMultiDbSearch::SetDatabasesToSearchL()`.

2.3.1.7.1 Configuring a search

`CPosLmMultiDbSearch` can be configured as `CPosLandmarkSearch` by setting certain parameters; see Section 2.3.1.1. The distinctions are:

- Setting a maximum number of matches limits the number of matches per database.
- If the client specifies to search only the previous matches and has changed which databases to search, new databases that were not a part of the previous search will generate no matches.

2.3.1.7.2 Specifying search criterion

With some restrictions, the criterion classes in Section 2.3.1.2 are used to specify what landmarks or landmark categories to search for. However, it is not allowed to:

- Search by category ID in `CPosLmCategoryCriteria` since an ID is only valid in one landmark database.
- Search with `CPosLmIdListCriteria` since an ID is only valid in one landmark database.

2.3.1.7.3 Starting a search

Starting a search is performed as described in Section 2.3.1.3.

One distinction is when executing `CPosLmOperation` in incremental steps. When searching multiple databases, this must be done asynchronously using an active object. It is not possible to execute `CPosLmOperation::NextStep()` synchronously using `User::WaitForRequest()`.

Another distinction is the error handling during a search. If an error occurs when executing a search operation from `CPosLandmarkSearch`, the search will terminate causing `CPosLmOperation::ExecuteL()` to leave and `CPosLmOperation::NextStep()` to complete with an error code. This is not the case for `CPosLmMultiDbSearch`.

For `CPosLmMultiDbSearch`, the search in the database where the error occurred will be terminated and the search will then continue in the remaining databases. This means that `CPosLmOperation::ExecuteL()` will never leave and `CPosLmOperation::NextStep()` will not complete with an error code (the only exception is when the client has an outstanding request when canceling the search, in which case the complete code will be `KErrCancel`). Instead, `CPosLmMultiDbSearch` must be checked for any errors encountered during the search; this is described in Section 2.3.1.7.5.

2.3.1.7.4 Checking incremental search progress and status

This is described in Section 2.3.1.4 but the status is not set to an error code if the operation has failed.

2.3.1.7.5 Checking for errors

The client can check the number of errors encountered during the search by calling `CPosLmMultiDbSearch::NumOfSearchErrors()`. Each error can be fetched by calling `CPosLmMultiDbSearch::GetSearchError()` and passing the index of the error.

2.3.1.7.6 Retrieving search matches from an iterator

This is described in Section 2.3.1.5.

For `CPosLmMultiDbSearch`, the IDs of the matches from the search can be retrieved per database so the client must specify the index of the database.

2.3.1.7.7 Retrieving search matches from display data

This is described in Section 2.3.1.6.

The display data contains matches from all databases. Each display item contains the index of the database where the match was found.

2.3.1.7.8 Releasing landmark resources

The Landmarks subsystem uses ECom plug-ins, which provide the implementation for accessing landmark databases. ECom allocates resources that are not released when the plug-in is unloaded. These must be explicitly released by the client at shutdown. This is done by calling the global method `ReleaseLandmarkResources()` which has the same effect as `REComSession::FinalClose()`.

The most common way to release landmark resources is to call `ReleaseLandmarkResources()` last thing in the client's destructor. If this is not done, the client may receive an ALLOC panic.

2.3.2 Error handling

The Landmarks Search API uses the standard Symbian error reporting mechanism. In case of a serious error, panics are used. Otherwise, errors are reported through return codes or leaves.

2.3.2.1 Panic codes

The Landmarks Search API uses the same panic code category as the Landmarks API. The panic codes are documented in reference document [2].

2.3.3 Memory overhead

If there are several matches (more than 1000) in a search, a large amount of memory is needed to store the matches. It might therefore be a good idea to set a maximum number of matches (see Section 2.3.1.1.1) before starting the search. Note, however, that it is the first found matches that are retrieved. For example, if the maximum number of matches is set when searching in sorted order, the result can be without a match although its name is in the beginning of the sort order.

2.3.4 Extensions to the API

There are no extensions defined to the Landmarks Search API.

2.4 Example

This section contains several examples of how to use the Landmarks Search API. All the examples assume that the client already has a handle to an open database; that is, a `CPosLandmarkDatabase` instance. How to open a landmark database is described in reference document [2].

2.4.1 Synchronous search using iterator

This example shows how to perform a search synchronously (not incrementally) in a landmark database. The matches are retrieved from an iterator.

```
// Create a search object and provide the CPosLandmarkDatabase
// object.
CPosLandmarkSearch* search = CPosLandmarkSearch::NewL(*database);
CleanupStack::PushL(search);

// Create the search criterion
_LIT(KSearchString, "flowers");
CPosLmTextCriteria* crit = CPosLmTextCriteria::NewLC();
crit->SetTextL(KSearchString);

// Start the search and execute it at once.
ExecuteAndDeleteLD(search->StartLandmarkSearchL(*crit));

CleanupStack::PopAndDestroy(crit);

// Retrieve an iterator to access the matching landmarks.
CPosLmItemIterator* iter = search->MatchIteratorL();
CleanupStack::PushL(iter);

// Iterate the search matches.
TPosLmItemId lmID;
while ((lmID = iter->NextL()) != KPosLmNullItemId)
{
    CPosLandmark* lm = database->ReadLandmarkLC(lmID);
    // Do something with the landmark information

    CleanupStack::PopAndDestroy(lm);
}

CleanupStack::PopAndDestroy(2, search);
```

2.4.2 Synchronous search using display data

This example shows how to perform a search synchronously (not incrementally) in a landmark database using display data.

```
// Create a search object and provide the CPosLandmarkDatabase
// object.
CPosLandmarkSearch* search = CPosLandmarkSearch::NewL(*database);
CleanupStack::PushL(search);

// Create a display data object
CPosLmDisplayData* displayData = CPosLmDisplayData::NewL();
CleanupStack::PushL(displayData);

// Set the display data to the search object.
search->SetDisplayData(*displayData);
```

```

// Create the search criterion
_LIT(KSearchString, "flowers");
CPosLmTextCriteria* crit = CPosLmTextCriteria::NewLC();
crit->SetTextL(KSearchString);

// Start the search and execute it all at once.
ExecuteAndDeleteLD(search->StartLandmarkSearchL(*crit));

CleanupStack::PopAndDestroy(crit);

// Iterate the search matches.
for (TInt i = 0; i < displayData->Count(); i++)
{
    const CPosLandmark& lm = displayData->DisplayItem(i).Landmark();
    // Do something with the landmark information
}

search->UnsetDisplayData();
CleanupStack::PopAndDestroy(2, search);

```

2.4.3 Asynchronous search

This example shows how to search a landmark database incrementally. The client also requests to sort the matches.

CSearchHandler class definition:

```

class CSearchHandler : public CActive
{
public:    // Constructors and destructor

    /**
     * Two-phased constructor.
     */
    static CSearchHandler* NewL(CPosLandmarkDatabase* aDb);

    /**
     * Destructor.
     */
    virtual ~CSearchHandler();

public:

    void StartLandmarkSearchL(CPosLmSearchCriteria* aCriteria);
    void StartCategorySearchL(CPosLmSearchCriteria* aCriteria);
    void NextSearchStep();
    void CleanupSearch();

    // From CActive
    void RunL();
    void DoCancel();
    TInt RunError(TInt aError);

private:

    /**

```

```

    * C++ default constructor.
    */
    CSearchHandler(CPosLandmarkDatabase* aDb);

    /**
    * By default Symbian 2nd phase constructor is private.
    */
    void ConstructL();

private:    // Data
    CPosLandmarkDatabase* iDb;
    CPosLandmarkSearch* iSearch;
    CPosLmDisplayData* iDisplayData;
    CPosLmOperation* iSearchOperation;
    TReal32 iProgress;
    TBool iIsSearchingForLandmarks;
};

```

CSearchHandler implementation:

```

// C++ default constructor
CSearchHandler::CSearchHandler(CPosLandmarkDatabase* aDb)
: CActive(EPriorityNormal), iDb(aDb)
{
}

// Symbian 2nd phase constructor can leave.
void CSearchHandler::ConstructL()
{
    iSearch = CPosLandmarkSearch::NewL(*iDb);

    iDisplayData = CPosLmDisplayData::NewL();
    iSearch->SetDisplayData(*iDisplayData);
}

// Two-phased constructor.
CSearchHandler* CSearchHandler::NewL(CPosLandmarkDatabase* aDb)
{
    CSearchHandler* self = new (ELeave) CSearchHandler(aDb);
    CleanupStack::PushL(self);
    self->ConstructL();
    CleanupStack::Pop();
    return self;
}

// Destructor
CSearchHandler::~CSearchHandler()
{
    Cancel();
    iSearch->UnsetDisplayData();
    delete iDisplayData;
    delete iSearch;
}

void CSearchHandler::StartLandmarkSearchL(CPosLmSearchCriteria*
aCriteria)
{

```

```

    TPosLmSortPref sp(CPosLandmark::ELandmarkName,
    TPosLmSortPref::EAscending);
    iSearchOperation = iSearch-
>StartLandmarkSearchL(*aCriteria, sp);
    iIsSearchingForLandmarks= ETrue;

    // Perform the first step in the incremental search operation.
    NextSearchStep();
}

void CSearchHandler::StartCategorySearchL(CPosLmSearchCriteria*
aCriteria)
{
    iSearch->StartCategorySearchL(*aCriteria,
        CPosLmCategoryManager::ECategorySortOrderNameAscending);
    iIsSearchingForLandmarks= EFalse;

    // Perform the first step in the incremental search operation.
    NextSearchStep();
}

void CSearchHandler::NextSearchStep()
{
    iSearchOperation->NextStep(iStatus, iProgress);
    SetActive();
}

void CSearchHandler::CleanupSearch()
{
    // Delete the search operation. This will cancel the
    // operation if it is not already complete.
    delete iSearchOperation;
    iSearchOperation = NULL;
}

void CSearchHandler::RunL()
{
    // Get all new matches since last step.
    TInt newItemIndex = iDisplayData->NewItemIndex();
    while (newItemIndex != KPosLmNoNewItems)
    {
        CPosLmDisplayItem& item = iDisplayData->DisplayItem(newItemIndex);
        if (iIsSearchingForLandmarks)
        {
            const CPosLandmark& lm = item.Landmark();
            // Do something with the landmark information
        }
        else
        {
            const CPosLandmarkCategory& category = item.Category();
            // Do something with the landmark category information
        }

        newItemIndex = iDisplayData->NewItemIndex();
    }

    if (iStatus == KPosLmOperationNotComplete)
    { // The search operation has not completed.
        // Use value iProgress to show progress bar to the
        // application user.
    }
}

```

```

        // Perform the next search step
        NextSearchStep();
    }
    else
    { // The search operation has completed.
        User::LeaveIfError(iStatus.Int());

        CleanupSearch();
    }
}

void CSearchHandler::DoCancel()
{
    CleanupSearch();
}

TInt CSearchHandler::RunError(TInt aError)
{
    // Notify application user of error and cleanup.
    CleanupSearch();
    return KErrNone;
}

```

2.4.4 Using composite criteria

Criteria can be combined using `CPosLmCompositeCriteria`. This example shows how to search for restaurants, which contain the text "thai":

```

// Create the composite criterion
CPosLmCompositeCriteria* compCrit = CPosLmCompositeCriteria::NewLC(
    CPosLmCompositeCriteria::ECompositionAND);

// Create the category search criterion and add it to composite
_LIT(KCategoryName, "restaurant");
CPosLmCategoryCriteria* catCrit =
CPosLmCategoryCriteria::NewLC();
catCrit->SetCategoryNameL(KCategoryName);

User::LeaveIfError(compCrit->AddArgument(catCrit));
// Ownership of the category criterion has been passed to the
// composite
CleanupStack::Pop(catCrit);

// Create the text search criterion and add it to composite
_LIT(KSearchString, "thai");
CPosLmTextCriteria* textCrit = CPosLmTextCriteria::NewLC();
textCrit->SetTextL(KSearchString);

User::LeaveIfError(compCrit->AddArgument(textCrit));
// Ownership of the text criterion has been passed to the
// composite
CleanupStack::Pop(textCrit);

// Start the search
ExecuteAndDeleteLD(search->StartLandmarkSearchL(*compCrit));
CleanupStack::PopAndDestroy(compCrit);

// Retrieve matches as usual.

```

2.4.5 Synchronous search in multiple databases

This example shows how to perform a search synchronously (not incrementally) in multiple landmark databases.

```
// Create a multi search object and provide a list of database URIs.
CPosLmMultiDbSearch* search = CPosLmMultiDbSearch::NewL(*aDatabaseList);
CleanupStack::PushL(search);

// Create a display data object.
CPosLmDisplayData* displayData = CPosLmDisplayData::NewL();
CleanupStack::PushL(displayData);

// Set the display data to the search object.
search->SetDisplayData(*displayData);

// Create the search criterion
_LIT(KSearchString, "flowers");
CPosLmTextCriteria* crit = CPosLmTextCriteria::NewLC();
crit->SetTextL(KSearchString);

// Start the search and execute it all at once.
ExecuteAndDeleteLD(search->StartLandmarkSearchL(*crit));
CleanupStack::PopAndDestroy(crit);

// Check if any errors have occurred.
TUint numErrors = search->NumOfSearchErrors();
for (TUint i = 0; i < numErrors; i++)
{
    CPosLmMultiDbSearch::TSearchError searchError;
    search->GetSearchError(i, searchError);
    // Do something with error
}

// Iterate the search matches.
for (TInt i = 0; i < displayData->Count(); i++)
{
    const CPosLandmark& lm = displayData->DisplayItem(i).Landmark();
    // Do something with the landmark information
}

search->UnsetDisplayData();
CleanupStack::PopAndDestroy(2, search);
```

2.4.6 Asynchronous search in multiple databases

This example shows how to search multiple landmark databases incrementally. The client also requests to sort the matches.

CMultiSearchHandler class definition:

```
class CMultiSearchHandler : public CActive
{
public: // Constructors and destructor

    /**
     * Two-phased constructor.
     */
    static CMultiSearchHandler* NewL(CDesCArray* aDatabaseList);
```

```

    /**
    * Destructor.
    */
    virtual ~CMultiSearchHandler();

public:

    void StartLandmarkSearchL(CPosLmSearchCriteria* aCriteria);

    void StartCategorySearchL(CPosLmSearchCriteria* aCriteria);

    void NextSearchStep();

    void CleanupSearch();

    // From CActive
    void RunL();
    void DoCancel();
    TInt RunError(TInt aError);

private:

    /**
    * C++ default constructor.
    */
    CMultiSearchHandler();

    /**
    * By default Symbian 2nd phase constructor is private.
    */
    void ConstructL(CDesCArray* aDatabaseList);

private:    // Data
    CPosLmMultiDbSearch* iSearch;
    CPosLmDisplayData* iDisplayData;
    CPosLmOperation* iSearchOperation;
    TReal32 iProgress;
    TBool iIsSearchingForLandmarks;
};

```

CMultiSearchHandler implementation:

```

// C++ default constructor
CMultiSearchHandler::CMultiSearchHandler()
: CActive(EPriorityNormal)
{
}

// Symbian 2nd phase constructor can leave.
void CMultiSearchHandler::ConstructL(CDesCArray* aDatabaseList)
{
    iSearch = CPosLmMultiDbSearch::NewL(*aDatabaseList);

    iDisplayData = CPosLmDisplayData::NewL();
    iSearch->SetDisplayData(*iDisplayData);
}

// Two-phased constructor.

```

```

CMultiSearchHandler* CMultiSearchHandler::NewL(CDesCArray*
aDatabaseList)
{
    CMultiSearchHandler* self = new (ELeave) CMultiSearchHandler();
    CleanupStack::PushL(self);
    self->ConstructL(aDatabaseList);
    CleanupStack::Pop();
    return self;
}

// Destructor
CMultiSearchHandler::~~CMultiSearchHandler()
{
    Cancel();
    iSearch->UnsetDisplayData();
    delete iDisplayData;
    delete iSearch;
}

void CMultiSearchHandler::StartLandmarkSearchL(CPosLmSearchCriteria*
aCriteria)
{
    TPosLmSortPref sp(CPosLandmark::ELandmarkName,
TPosLmSortPref::EAscending);
    iSearchOperation = iSearch->StartLandmarkSearchL(*aCriteria, sp);
    iIsSearchingForLandmarks= ETrue;

    // Perform the first step in the incremental search operation.
    NextSearchStep();
}

void CMultiSearchHandler::StartCategorySearchL(CPosLmSearchCriteria*
aCriteria)
{
    iSearch->StartCategorySearchL(*aCriteria,
        CPosLmCategoryManager::ECategorySortOrderNameAscending);
    iIsSearchingForLandmarks= EFalse;

    // Perform the first step in the incremental search operation.
    NextSearchStep();
}

void CMultiSearchHandler::NextSearchStep()
{
    iSearchOperation->NextStep(iStatus, iProgress);
    SetActive();
}

void CMultiSearchHandler::CleanupSearch()
{
    // Delete the search operation. This will cancel the
    // operation if it is not already complete.
    delete iSearchOperation;
    iSearchOperation = NULL;
}

void CMultiSearchHandler::RunL()
{
    // Get all new matches since last step.
    TInt newItemIndex = iDisplayData->NewItemIndex();

```

```

while (newItemIndex != KPosLmNoNewItems)
{
    CPosLmDisplayItem& item = iDisplayData->DisplayItem(newItemIndex);
    if (iIsSearchingForLandmarks)
    {
        const CPosLandmark& lm = item.Landmark();
        // Do something with the landmark information
    }
    else
    {
        const CPosLandmarkCategory& category = item.Category();
        // Do something with the landmark category information
    }

    newItemIndex = iDisplayData->NewItemIndex();
}

if (iStatus == KPosLmOperationNotComplete)
{ // The search operation has not completed.
  // Use value iProgress to show progress bar to the
  // application user.

  // Perform the next search step
  NextSearchStep();
}
else
{ // The search operation has completed.
  // Check if any errors have occurred.
  TUint numErrors = iSearch->NumOfSearchErrors();
  for (TUint i = 0; i < numErrors; i++)
  {
      CPosLmMultiDbSearch::TSearchError searchError;
      iSearch->GetSearchError(i, searchError);
      // Do something with error
  }

  CleanupSearch();
}
}

void CMultiSearchHandler::DoCancel()
{
    CleanupSearch();
}

TInt CMultiSearchHandler::RunError(TInt aError)
{
    // Notify application user of error and cleanup.
    CleanupSearch();
    return KErrNone;
}

```

2.5 Detailed description

The Landmarks Search API classes are documented in detail in the following sections.

2.5.1 CPosLandmarkSearch

This class is used to perform searches for landmarks or landmark categories in a landmark database.

To search a landmark database, an instance of this class is created, supplying the database to search. The client creates a criterion object to specify what to search for. There are different criterion classes which all inherit from `CPosLmSearchCriteria`. For instance, the client can search for landmarks that contain a certain text string by passing a `CPosLmTextCriteria`.

Some criterion classes are used for searching for landmarks (for example, `CPosLmCategoryCriteria` is used for searching for landmarks that contain a certain category) and some are used to search for landmark categories (for example, `CPosLmCatNameCriteria` is used to search for landmark categories by specifying a category name which may contain wild card characters).

Searches can be run in incremental mode. `StartLandmarkSearchL` and `StartCategorySearchL` both return a `CPosLmOperation` object, which is used to execute the search. If it is run incrementally, the client can supervise the progress of the operation. If it is sufficient to run the search synchronously, the client only has to call `CPosLmOperation::ExecuteL`. The client can cancel the search by deleting the `CPosLmOperation` object.

By default, these functions start a new search, but the client can specify that only the previous matches should be searched. This can be used to refine the search when there are many matches.

During the search execution, a read-lock is acquired for each database.

Only one search can be performed at a time by the single instance of the class. If a search is already running, the search function leaves with the error code `KErrInUse`.

After search completion, the client can make a second search and specify that only the matches from the previous search shall be considered.

The client can also set a limit on how many search matches should be returned (see `SetMaxNumOfMatches` in Section 2.5.1.4).

The client retrieves the matches from the search by requesting an iterator (see `MatchIteratorL` in Section 2.5.1.5) or by using display data (see `SetDisplayData` in Section 2.5.1.4).

The `NetworkServices` capability is required for remote databases.

2.5.1.1 Construction

```
static CPosLandmarkSearch* NewL(CPosLandmarkDatabase&
aDatabase)
```

Usage:

This is a two-phased constructor. The client takes ownership of the returned search object.

Parameters:

Parameter	Description
aDatabase	The landmark database to search.

Returns:

A new instance of this class.

2.5.1.2 Starting a search for landmarks

```
virtual CPosLmOperation* StartLandmarkSearchL(
    const CPosLmSearchCriteria& aCriteria,
    const TPosLmSortPref& aSortPref,
    TBool aSearchOnlyPreviousMatches = EFalse) [pure virtual]
```

Usage:

Starts a search for landmarks.

This overload of the `StartLandmarkSearchL` function lets the client define the sort order for the search matches.

Only sorting by landmark name is supported. If the client tries to sort by another attribute, this function will leave with the error code `KErrNotSupported`.

If there are no previous matches and the client specifies that previous matches should be used, this function leaves with the error code `KErrArgument`.

If a search is already running, this function leaves with the error code `KErrInUse`.

If the search criterion is not valid for landmark searching, this function leaves with the error code `KErrArgument`.

If the search criterion is not supported, this function leaves with the error code `KErrNotSupported`.

The client takes ownership of the returned operation object.

This function requires the `ReadUserData` capability.

Parameters:

Parameter	Description
aCriteria	The search criterion.
aSortPref	A sort preference object.
aSearchOnlyPreviousMatches	This flag may be used to perform a search within the results of previous search.

Returns:

A handle to the search operation.

```
virtual CPosLmOperation* StartLandmarkSearchL(
    const CPosLmSearchCriteria& aCriteria,
    TBool aSearchOnlyPreviousMatches = EFalse) [pure virtual]
```

Usage:

Starts a search for landmarks.

If there are no previous matches and the client specifies that previous matches should be used, this function leaves with the error code `KErrArgument`.

If a search is already running, this function leaves with the error code `KErrInUse`.

If the search criterion is not valid for landmark searching, this function leaves with the error code `KErrArgument`.

If the search criterion is not supported, this function leaves with the error code `KErrNotSupported`.

The client takes ownership of the returned operation object.

This function requires the `ReadUserData` capability.

Parameters:

Parameter	Description
<code>aCriteria</code>	The search criterion.
<code>aSearchOnlyPreviousMatches</code>	This flag may be used to perform a search within the results of previous search.

Returns:

A handle to the search operation.

2.5.1.3 Starting a search for categories

```
virtual CPosLmOperation* StartCategorySearchL(
    const CPosLmSearchCriteria& aCriteria,
    CPosLmCategoryManager::TCategorySortPref aSortPref,
    TBool aSearchOnlyPreviousMatches = EFalse) [pure virtual]
```

Usage:

Starts a search for landmark categories.

The criterion defining whether a landmark category is a match or not is passed as input to this function.

If a search is already running, this function leaves with the error code `KErrInUse`.

If the search criterion is not valid for landmark category searching, this function leaves with the error code `KErrArgument`.

If the search criterion is not supported, this function leaves with the error code `KErrNotSupported`.

The client takes ownership of the returned operation object.

This function requires the `ReadUserData` capability.

Parameters:

Parameter	Description
<code>aCriteria</code>	The search criterion.
<code>aSortPref</code>	Sort preference for the search results.
<code>aSearchOnlyPreviousMatches</code>	This flag may be used to perform a search within the results of previous search.

Returns:

A handle to the search operation.

2.5.1.4 Options

```
TInt MaxNumOfMatches() const
```

Usage:

Retrieves the maximum number of search matches limit.

By default, the maximum number of matches is unlimited.

Returns:

The maximum number of search matches or `KPosLmMaxNumOfMatchesUnlimited` if the number of search matches is unlimited.

```
void SetMaxNumOfMatches(TInt aMaxNumOfMatches =
KPosLmMaxNumOfMatchesUnlimited)
```

Usage:

Sets the maximum number of search matches limit.

If the limit is set, the search operation will stop when this limit is reached.

By default, the maximum number of matches is unlimited.

If a new value for the maximum number of matches is set when a search is ongoing, it will not affect the current search. The new maximum will be utilized in the next search.

Parameters:

Parameter	Description
<code>aMaxNumOfMatches</code>	The maximum number of search matches. <code>KPosLmMaxNumOfMatchesUnlimited</code> means that the number of matches is unlimited.

```
virtual void SetDisplayData(CPosLmDisplayData& aData) [pure virtual]
```

Usage:

Display data can be used as an alternative way to get results from a database search. Landmarks or categories are added to the display data collection during a search depending on the search type.

This function may replace the combination of using `MatchIteratorL` and reading landmark or category data. Result data is read already during the search and no duplicate access to the database is needed.

The display data object will be reset each time a new search is started. No items during the search are removed from the collection. New found matches can be added every time a next search step is completed, see

`CPosLmDisplayData::NewItemIndex` in Section 2.5.10.3.



Note: The database index of the displayable data items in `CPosLmDisplayData` will be set to 0 as it has no meaning in a single database search.

If the client sets display data during an ongoing search, this function panics with `EPosSearchOperationInUse`.

The client owns the display data object. If the client deletes it during a search, this may lead to unexpected errors. The client must call `UnsetDisplayData` before it deletes the display data object.



Note: If the client resets the display data during a search, the sort order in the iterator might become incorrect.

Parameters:

Parameter	Description
<code>aData</code>	The displayable data.

```
virtual void UnsetDisplayData() [pure virtual]
```

Usage:

Unsets display data. No further data will be added to the display data that was set with `SetDisplayData`.

If the client unsets display data during an ongoing search, this function panics with `EPosSearchOperationInUse`.

2.5.1.5 Retrieving results

```
virtual TUint NumOfMatches() const [pure virtual]
```

Usage:

Returns the number of matches so far in the search.

This function can also be called during a search operation.

Returns:

The number of search matches.

```
virtual CPosLmItemIterator* MatchIteratorL() [pure virtual]
```

Usage:

Creates an iterator object to iterate the matching landmarks or landmark categories.

This function can also be called during a search in order to read the matches encountered so far. Note that the iterator will not iterate any new matches. If new matches are found, a new iterator must be created.

If a sort preference was specified when the search was started, the landmarks/categories will be sorted when the search is complete but the items are not necessarily sorted if this function is called during a search.



Note: If the client has set display data and resets the display data during a search, the sort order in the iterator might be incorrect.

The client takes ownership of the returned iterator object.



Note: The iterator iterates matches in `CPosLandmarkSearch`. It cannot be used after the search object has been deleted. Make sure to delete the iterator first.

Returns:

A search match iterator.

2.5.2 CPosLmSearchCriteria

`CPosLmSearchCriteria` is an abstract base class for landmark search criterion classes.

Criterion classes are used in `CPosLandmarkSearch` to specify what to search for. It specifies what criteria the landmark must fulfill in order to be considered a search match.

2.5.2.1 CriteriaType

```
enum TCriteriaType
```

Usage:

Specifies the subclass of the criterion object.

Enumeration values:

Value	Description
<code>ECriteriaArea</code>	The subclass <code>CPosLmAreaCriteria</code>
<code>ECriteriaText</code>	The subclass <code>CPosLmTextCriteria</code>

Value	Description
ECriteriaComposite	The subclass CPosLmCompositeCriteria
ECriteriaCategory	The subclass CPosLmCategoryCriteria
ECriteriaFindNearest	The subclass CPosLmNearestCriteria
ECriteriaIdList	The subclass CPosLmIdListCriteria
ECriteriaCategoryByName	The subclass CPosLmCatNameCriteria

```
TCriteriaType CriteriaType() const
```

Usage:

Returns the criterion type.

Returns:

The criterion type.

2.5.3 CPosLmTextCriteria

CPosLmTextCriteria contains criterion for searching for landmarks that contain a certain text.

The search is defined by providing a text to search for and the position fields and text attributes to search.

If no attributes are specified for the search, all text attributes in the landmarks are searched. If no position fields are specified, then all fields are searched.

If this criterion is passed to CPosLandmarkSearch::StartLandmarkSearchL, only landmarks that contain the specified text are returned.

Wild card characters are supported.

This criterion is only valid when searching for landmarks; that is, if it is passed to CPosLandmarkSearch::StartCategorySearchL, the function will fail with the error code KErrArgument.

2.5.3.1 Construction

```
static CPosLmTextCriteria* NewLC()
```

Usage:

A two-phased constructor.

Returns:

A new instance of this class.

2.5.3.2 Search text

```
void SetTextL(const TDesC& aText)
```

Usage:

Sets the search string.

A non-empty text string must be set; otherwise

`CPosLandmarkSearch::StartLandmarkSearchL` will leave with the error code `KErrArgument`.

The search is case insensitive.

Wild card characters "?" and "*" are supported in the search string. "?" matches a single occurrence of any character and "*" matches zero or more consecutive occurrences of any characters.

A landmark matches the criterion if the specified text is found anywhere in the selected attributes or position fields.

If the search string is longer than `KPosLmMaxSearchStringLength`, this function will leave with the error code `KErrArgument`.

Parameters:

Parameter	Description
<code>aText</code>	The text to search for.

```
TPtrC Text() const
```

Usage:

Retrieves the text to search for.

Returns:

The text to search for.

2.5.3.3 Attributes to search

```
void SetAttributesToSearch(CPosLandmark::TAttributes  
aAttributes)
```

Usage:

Sets which landmark attributes to search.

The client passes a bitmap of the landmark attributes to search. It is only possible to search the `ELandmarkName` and `EDescription` attributes. If any other attribute is passed to this function, it panics with the code `EPosInvalidLandmarkAttribute`.

Parameters:

Parameter	Description
aAttributes	A bitmap indicating which landmark attributes should be searched.

```
CPosLandmark::TAttributes AttributesToSearch() const
```

Usage:

Retrieves the landmark attributes to be searched.

This function returns a bitmap of the landmark attributes. The landmark attributes are defined by `CPosLandmark::_TAttributes`. It is only possible to search the `ELandmarkName` and `EDescription` attributes.

Returns:

A bitmap indicating which landmark attributes should be searched.

2.5.3.4 Fields to search

```
void SetPositionFieldsToSearchL(const RArray<TUint>&
aFieldArray)
```

Usage:

Sets the position fields to search.

Any previously set position fields are cleared by this call.

Parameters:

Parameter	Description
aFieldArray	A list of the position fields to search. Position fields are identified by values defined in <code>TPositionFieldId</code> .

```
void ClearPositionFieldsToSearch()
```

Usage:

Clears the position fields list used in a search.

```
void GetPositionFieldsToSearchL(RArray<TUint>& aFieldArray)
const
```

Usage:

Retrieves a list of the position fields to search.

Parameters:

Parameter	Description
aFieldArray	On return, contains a list of the position fields, which are used in a search. If no position fields have been set, the array is empty. Position fields are identified by values defined in TPositionFieldId.

2.5.4 CPosLmCategoryCriteria*2.5.4.1 Description*

CPosLmCategoryCriteria contains criterion used for searching for landmarks that belong to a certain category.

A category has an ID in the database and it has also a unique name. There are also some global categories that are known by all databases. It is possible to specify either a local category in a specific landmark database, a global category or the name of the category. If one is set, the others are reset and not used.

It is also possible to search for uncategorized landmarks; that is, landmarks that do not contain any categories. If neither item ID, global ID, nor name are set, CPosLandmarkSearch::StartLandmarkSearchL will search for uncategorized landmarks.

This criterion is only valid when searching for landmarks; that is, if it is passed to CPosLandmarkSearch::StartCategorySearchL, the function will fail with the error code KErrArgument.

2.5.4.2 Construction

```
static CPosLmCategoryCriteria* NewLC()
```

Usage:

A two-phased constructor.

Returns:

A new instance of this class.

2.5.4.3 Category name

```
void SetCategoryNameL(const TDesC& aCategoryName)
```

Usage:

Sets the name of the category which should be used as the landmark search criterion.

The exact category name must be specified. Wild card characters are not used as such; they will be considered as ordinary characters. It is possible to use CPosLmCatNameCriteria to list categories that match a category name pattern containing wild card characters.

The matching is case sensitive.

An empty descriptor means uncategorized landmarks search.

If the category name is longer than `KPosLmMaxCategoryNameLength`, this function will leave with the error code `KErrArgument`.

Parameters:

Parameter	Description
<code>aCategoryName</code>	The category name.

```
TPtrC CategoryName() const
```

Usage:

Returns the name of the category which should be used as the landmark search criterion.

Returns:

The category name or an empty descriptor if not set.

2.5.4.4 *Category item*

```
void SetCategoryId(TPosLmItemId aItemId)
```

Usage:

Sets the item ID of the category which should be used as the landmark search criterion.

Passing `KPosLmNullItemId` means uncategorized landmarks search.

Parameters:

Parameter	Description
<code>aItemId</code>	The item ID of the category.

```
TPosLmItemId CategoryItemId() const
```

Usage:

Returns the item ID of the category which should be used as the landmark search criterion.

Returns:

The item ID of the category or `KPosLmNullItemId` if the ID is not set.

2.5.4.5 *Global category*

```
void SetGlobalCategory(TPosLmGlobalCategory aGlobalCategory)
```

Usage:

Sets the global category which should be used as the landmark search criterion.

Passing `KPosLmNullGlobalCategory` means uncategorized landmarks search.

Parameters:

Parameter	Description
aGlobalCategory	The global category.

```
TPosLmGlobalCategory GlobalCategory() const
```

Usage:

Returns the global category which should be used as the landmark search criterion.

Returns:

The global category or `KPosLmNullGlobalCategory` if the global category is not set.

2.5.5 CPosLmCatNameCriteria

`CPosLmCatNameCriteria` contains criterion for searching landmark categories with a certain name.

Wild card characters are supported.

The client specifies the search pattern and starts the search using `CPosLandmarkSearch`. The search returns all the categories that match the search pattern.

This criterion is only valid when searching for landmark categories; that is, if it is passed to `CPosLandmarkSearch::StartLandmarkSearchL`, the function will fail with the error code `KErrArgument`.

2.5.5.1 Construction

```
static CPosLmCatNameCriteria* NewLC()
```

Usage:

A two-phased constructor.

Returns:

A new instance of this class.

2.5.5.2 Search pattern

```
TPtrC SearchPattern() const
```

Usage:

Retrieves the search pattern.

Returns:

The category name search pattern.

```
void SetSearchPatternL(const TDesC& aSearchPattern)
```

Usage:

Sets the search pattern.

A non-empty search pattern must be set; otherwise `CPosLandmarkSearch::StartCategorySearchL` will leave with the error code `KErrArgument`.

The search is case insensitive.

Wild card characters "?" and "*" are supported in the search string. "?" matches a single occurrence of any character and "*" matches zero or more consecutive occurrences of any characters.

If the search pattern is longer than `KPosLmMaxCategoryNameLength`, this function will leave with the error code `KErrArgument`.

Parameters:

Parameter	Description
<code>aSearchPattern</code>	The pattern used to find categories.

2.5.6 CPosLmAreaCriteria

`CPosLmAreaCriteria` contains criterion for searching for landmarks that reside in a certain area.

The search area is defined by providing two latitude and two longitude values that specify the borders of the area. Note that this search does not consider landmark coverage radius; see `CPosLandmark::GetCoverageRadius` in reference document [2].

The search area is defined as a spherical rectangle, limited by two longitude borders (`WestLongitude` and `EastLongitude`) and two latitude borders (`SouthLatitude` and `NorthLatitude`).

It is required that $-90 \leq \text{SouthLatitude} \leq \text{NorthLatitude} \leq 90$. `WestLongitude` must be in the interval $[-180, 180)$, that is, not including +180. `EastLongitude` must be in the interval $[-180, 180]$, that is, including +180. It is allowed that `EastLongitude` is smaller than `WestLongitude`. This defines an area that crosses the 180 meridian. The area definition is inclusive; that is, landmarks that lie on the border of the area will be considered as matches. If `WestLongitude` equals `EastLongitude` and `NorthLatitude` equals `SouthLatitude`, then only the landmarks that match the longitude and latitude respectively are considered to be matches. If `WestLongitude` = -180 and `EastLongitude` = +180, all longitudes are included in the search.

This criterion is only valid when searching for landmarks; that is, if it is passed to `CPosLandmarkSearch::StartCategorySearchL`, the function will fail with the error code `KErrArgument`.

2.5.6.1 Functions

```
static CPosLmAreaCriteria* NewLC(
    const TReal64& aSouthLatitude,
    const TReal64& aNorthLatitude,
```

```

    const TReal64& aWestLongitude,
    const TReal64& aEastLongitude)

TInt SetSearchArea(
    const TReal64& aSouthLatitude,
    const TReal64& aNorthLatitude,
    const TReal64& aWestLongitude,
    const TReal64& aEastLongitude)

void GetSearchArea(
    TReal64& aSouthLatitude,
    TReal64& aNorthLatitude,
    TReal64& aWestLongitude,
    TReal64& aEastLongitude) const

```

Usage:

The input parameters specify the area to search as described in Section 2.5.6.

If one of the parameters is out of range, this function fails with the error code `KErrArgument`. The correct ranges are:

1. `-90 <= aSouthLatitude <= aNorthLatitude <= 90`
2. `-180 <= aWestLongitude < 180`
3. `-180 <= aEastLongitude <= 180`

The `GetSearchArea` function returns the current search area parameters.

Parameters:

Parameter	Description
<code>aSouthLatitude</code>	The southern latitude border of the search area.
<code>aNorthLatitude</code>	The northern latitude border of the search area.
<code>aWestLongitude</code>	The western longitude border of the search area.
<code>aEastLongitude</code>	The eastern longitude border of the search area.

Returns:

`NewLC` returns a new instance of this class.

`SetSearchArea` returns `KErrNone` if successful, `KErrArgument` if the input parameters are invalid, otherwise a system wide error code.

2.5.7 CPosLmNearestCriteria

`CPosLmNearestCriteria` contains criterion for finding the landmarks that are closest to a certain coordinate, called "center coordinate".

When using `CPosLmNearestCriteria`, the matches returned in the search are sorted in ascending distance order if the client does not specify other sort preferences.

By default, this search returns all the landmarks in the database, except for those, which have no coordinates. It is recommended to specify a maximum distance to narrow down the search. This is done by `SetMaxDistance`.

The client can specify if the coverage radius should be used. If coverage radius is used, the distance to the landmark coverage area border is used instead of the distance to the landmark coverage area center. If the source coordinate is inside the landmark coverage area, the distance is considered as zero. By default, the coverage radius is not used.

This criterion is only valid when searching for landmarks; that is, if it is passed to `CPosLandmarkSearch::StartCategorySearchL`, the function will fail with the error code `KErrArgument`.

2.5.7.1 Construction

```
static CPosLmNearestCriteria* NewLC(
    const TCoordinate& aCoordinate,
    TBool aUseCoverageRadius = EFalse)
```

Usage:

A two-phased constructor.

Latitude and longitude must be set in the coordinate (not `NaN`); otherwise this function will panic with the code `EPosNaNCoordinate`. Altitude is ignored.

Parameters:

Parameter	Description
<code>aCoordinate</code>	The center coordinate of the landmark search.
<code>aUseCoverageRadius</code>	<code>ETrue</code> if coverage radius should be considered in the search.

2.5.7.2 Center coordinate

```
void SetCoordinate(const TCoordinate& aCoordinate)
void GetCoordinate(TCoordinate& aCoordinate) const
```

Usage:

Sets and gets the center coordinate of the search.

Latitude and longitude must be set in the coordinate (not `NaN`); otherwise this function panics with the code `EPosNaNCoordinate`. Altitude is ignored.

Parameters:

Parameter	Description
<code>aCoordinate</code>	The center coordinate of the landmarks search.

2.5.7.3 Distance

```
void SetMaxDistance(TReal32 aMaxDistance)
TReal32 MaxDistance() const
```

Usage:

Sets and gets the maximum distance for the search.

Only the landmarks that are closer than the maximum distance are considered in the search.

By default, the maximum distance is set to `NaN`, which means that the distance is unlimited.

Parameters:

Parameter	Description
<code>aMaxDistance</code>	The maximum distance or <code>NaN</code> if the distance should be unlimited.

Returns:

The maximum distance or `NaN` if the distance is unlimited.

2.5.7.4 CoverageRadius

```
void SetUseCoverageRadius(TBool aUseCoverageRadius)
```

```
TBool UseCoverageRadius() const
```

Usage:

Sets and returns if coverage radius should be considered in the search.

Parameters:

Parameter	Description
<code>aUseCoverageRadius</code>	<code>ETrue</code> if coverage radius should be considered in the search. Otherwise <code>EFalse</code> .

Returns:

`ETrue` if coverage radius is considered in the search; otherwise `EFalse`.

2.5.8 CPosLmCompositeCriteria

`CPosLmCompositeCriteria` is a class used to combine multiple search criteria.

For instance, to search for all the restaurants in the area, this class can be used to combine `CPosLmAreaCriteria` and `CPosLmCategoryCriteria`.

If `CPosLmNearestCriteria` is used and no sort preference is specified, the result will be sorted by distance. If more than one `CPosLmNearestCriteria` are combined using `CPosLmCompositeCriteria`, the sort order will be undefined unless a sort preference is specified.

Nested `CPosLmCompositeCriteria` are not allowed.

This criterion only supports searching for landmarks; for example, if it is passed to `CPosLandmarkSearch::StartCategorySearchL`, the function will fail with the error code `KErrNotSupported`.

2.5.8.1 Construction

```
static CPosLmCompositeCriteria* NewLC(TCompositionType aType)
```

Usage:

A two-phased constructor.

Parameters:

Parameter	Description
aType	The composition type to use. Currently only ECompositionAND is supported.

Returns:

A new instance of this class.

2.5.8.2 Composition type

Types:

```
enum TCompositionType
```

Usage:

Specifies the type of the composite criterion.

Enumeration values:

Value	Description
ECompositionAND	Search results must match all the contained criteria.

Functions:

```
TCompositionType CompositionType() const
void SetCompositionType(TCompositionType aType)
```

Usage:

Gets and sets the type of this composite criterion.

Parameters:

Parameter	Description
aType	The composition type to use. Currently only ECompositionAND is supported.

Returns:

The composition type. Currently it is always ECompositionAND.

2.5.8.3 Arguments

```
TInt AddArgument(CPosLmSearchCriteria* aCriteria)
```

Usage:

Adds a criterion to this composition.

The composite criterion must contain at least one argument; otherwise `CPosLandmarkSearch::StartLandmarkSearchL` will fail with the error code `KErrArgument`.

If this function returns without an error code, the ownership of the added criterion object is transferred to the composite object.

If `CPosLmCompositeCriteria` is specified as argument to this function, it will return `KErrNotSupported`. Nested `CPosLmCompositeCriteria` are not supported.

If `CPosLmCatNameCriteria` is specified as argument to this function, it will return `KErrNotSupported`. Searching for landmark categories using `CPosLmCompositeCriteria` is not supported.

Parameters:

Parameter	Description
<code>aCriteria</code>	The criterion to add to the composition.

Returns:

`KErrNone` if the operation was successful; otherwise a system wide error code.

```
CPosLmSearchCriteria& Argument(TUint aIndex)
```

```
const CPosLmSearchCriteria& Argument(TUint aIndex) const
```

Usage:

Returns the criterion argument contained in this object.

The first overload of the `Argument` function returns a non-constant reference to the criterion argument.

The second overload of the `Argument` function returns a constant reference to the criterion argument.

Parameters:

Parameter	Description
<code>aIndex</code>	The argument to read. Must be in the interval <code>[0, NumOfArguments - 1]</code> , or this function will raise a USER-130 panic.

Returns:

The requested argument.

```
void ClearArguments()
```

Usage:

Removes and deletes all the contained criterion objects.

The composite criterion must contain at least one argument; otherwise `CPosLandmarkSearch::StartLandmarkSearchL` will fail with the error code `KErrArgument`.

```
TUint NumOfArguments() const
```

Usage:

Returns the number of criteria this object contains.

Returns:

The number of criteria this object contains.

```
CPosLmSearchCriteria* RemoveArgument (TUint aIndex)
```

Usage:

Removes a criterion from this composition.



Note: This function does not delete the criterion object. Instead, the ownership of the object is passed to the caller.

Parameters:

Parameter	Description
aIndex	The argument to remove. Must be in the interval [0, NumOfArguments - 1], or this function will raise a <code>USER-130</code> panic.

Returns:

The criterion object that was removed from the composition.

2.5.9 CPosLmIdListCriteria

`CPosLmIdListCriteria` contains Landmark ID list search criterion.

This criterion is used if the client only wants to search a subset of the landmarks in the database.

This criterion must be combined with other search criteria using `CPosLmCompositeCriteria`. It is of no use on its own. If it is not combined with another criterion, `CPosLandmarkSearch::StartLandmarkSearchL` will fail with the error code `KErrArgument`.

For example, if this criterion is combined with `CPosLmTextCriteria`, the search operation searches the landmarks specified in the ID list criterion and returns those that match the given text string.

Only one ID list criterion is allowed in each composite criterion; otherwise `CPosLandmarkSearch::StartLandmarkSearchL` will fail with the error code `KErrArgument`.

If the criterion does not contain any landmark IDs, `CPosLandmarkSearch::StartLandmarkSearchL` will fail with the error code `KErrArgument`.

2.5.9.1 Construction

```
static CPosLmIdListCriteria* NewLC()
```

Usage:

A two-phased constructor.

Returns:

A new instance of this class.

2.5.9.2 List of items

```
void GetLandmarkIdsL(RArray<TPosLmItemId>& aIdArray) const
```

Usage:

Retrieves a list of the IDs of the landmarks that should be included in the search.

Parameters:

Parameter	Description
<code>aIdArray</code>	On return, contains the IDs of the landmarks that should be included in the search.

```
void SetLandmarkIdsL(const RArray<TPosLmItemId>& aIdArray)
```

Usage:

Sets the IDs of the landmarks that should be included in the search.

Parameters:

Parameter	Description
<code>aIdArray</code>	The IDs of the landmarks that should be included in the search.

2.5.10 CPosLmDisplayData

CPosLmDisplayData contains a displayable data collection.

A displayable data collection consists of displayable items (CPosLmDisplayItem). A displayable data collection consists either of landmark items (if a landmark search has been started) or category items (if a category search has been started). Items of different types cannot be mixed in the collection.

Displayable data is used in CPosLandmarkSearch and CPosLmMultiDbSearch to hold search results. The collection is populated with new results every time the next search step is executed. Displayable items contain full or partial (see SetPartialReadParameters) landmark data or full category data, and can be used to display search results already during the search and also after it has completed.



Note: The single class instance may only be used by one search instance at a time.

2.5.10.1 Construction

```
static CPosLmDisplayData* NewL()
```

Usage:

A two-phased constructor.

Returns:

A new instance of this class.

2.5.10.2 Options

```
void SetPartialReadParametersL(
    const CPosLmPartialReadParameters& aPartialSettings)
```

Usage:

Sets the partial read parameters for this display data.

Partial read parameters are used to define which landmark data should be read during a landmark search. If no partial read parameters are set, the whole landmark will be read.

This function only affects searches that are started after the function is called. The current search is not affected.

If landmarks are sorted by name, the name will always be a part of the landmark in the display data, even if it is not requested.



Note: Partial read parameters are only used for landmark searches.

Parameters:

Parameter	Description
aPartialSettings	The partial read parameters.

```
void UnsetPartialReadParameters ()
```

Usage:

Unsets the partial read parameters for this display data.

This means that from now, on all the landmarks added to this display data instance will contain all the information.

To have any affect, this function must be called before a search is started. If it is called during a search, it will only affect the next search.



Note: Partial read parameters will only have effect on landmark searches.

```
void Reset ()
```

Usage:

Resets the collection and deletes all the contained items.

2.5.10.3 Retrieving results

```
TInt NewItemIndex ()
```

Usage:

Returns the index of the next match found during the current asynchronous search operation. If the search is executed synchronously, then all the indexes of the matches found during the search are returned one by one.

After each search step, the new item indexes are returned in ascending order according to the specified sort preference in the search.

Returns:

The item index or `KPosLmNoNewItems` when no new items are available.

```
TInt Count () const
```

Usage:

Returns the number of items in the collection.

Returns:

The number of items.

```
CPosLmDisplayItem& DisplayItem(TInt aItemIndex) const
```

Usage:

Returns the displayable item specified by index. The index must be strictly less than `Count` and not less than 0; otherwise this function panics with `EPosInvalidIndex`.

Parameters:

Parameter	Description
<code>aItemIndex</code>	The displayable item index.

Returns:

The displayable item.

2.5.11 CPosLmDisplayItem

`CPosLmDisplayItem` contains a displayable item.

A displayable item consists of a landmark or category and its database index. The database index is 0 if running a single database search, and it is in the range `[0, CPosLmMultiDbSearch::NumOfDatabasesInSearch - 1]` in case of a multiple database search. Thus, this class is a link between a landmark or a category and its database.

The class is usually not instantiated by client applications.

2.5.11.1 Construction

```
static CPosLmDisplayItem* NewL(
    CPosLandmark* aLandmark,
    TInt aDatabaseIndex = 0)
```

Usage:

A two-phased constructor.

Parameters:

Parameter	Description
<code>aLandmark</code>	A landmark.
<code>aDatabaseIndex</code>	A database index.

Returns:

A new instance of this class.

```
static CPosLmDisplayItem* NewL(
    CPosLandmarkCategory* aCategory,
    TInt aDatabaseIndex = 0)
```

Usage:

A two-phased constructor.

Parameters:

Parameter	Description
aCategory	A category.
aDatabaseIndex	A database index.

Returns:

A new instance of this class.

2.5.11.2 Display item type

```
enum TDisplayItemType
```

Usage:

The display item type.

Enumeration values:

Value	Description
ELandmarkItem	A landmark display item. This indicates that the item contains a landmark and the Landmark function can be called to access it.
ECategoryItem	A category display item. This indicates that the item contains a category and the Category function can be called to access it.

```
TDisplayItemType DisplayItemType() const
```

Usage:

Returns the type of the display item.

Returns:

The display item type.

2.5.11.3 Display item index

```
TUint DatabaseIndex() const
```

Usage:

Returns the index of the database to which the contained item belongs.

The database index is associated with a database URI from the list of databases specified in `CPosLmMultiDbSearch`.

If display data is used in `CPosLandmarkSearch`, this function always returns 0.

Returns:

The database index of this displayable item.

2.5.11.4 Category

```
const CPosLandmarkCategory& Category() const
```

Usage:

Returns the category contained in the displayable item.

This function panics with `EPosInvalidItemType` if the item type is not a category displayable item. See `DisplayItemType` in Section 2.5.11.2.

Returns:

The category.

2.5.11.5 Landmark

```
const CPosLandmark& Landmark() const
```

Usage:

Returns the landmark contained in the displayable item.

This function panics with `EPosInvalidItemType` if the item type is not a landmark displayable item. See `DisplayItemType` in Section 2.5.11.2.

Returns:

The landmark.

2.5.11.6 Distance

```
TInt GetDistance(TReal32& aDistance) const
```

Usage:

Returns the distance to a position specified in `CPosLmNearestCriteria`. The distance data is only used when searching with this criterion.

Parameters:

Parameter	Description
<code>aDistance</code>	The distance to the position specified in <code>CPosLmNearestCriteria</code> .

Returns:

`KErrNone` if the distance is available; otherwise `KErrNotFound`.

2.5.12 CPosLmMultiDbSearch

`CPosLmMultiDbSearch` is used to perform searches for landmarks or landmark categories in multiple databases.

The client can specify which databases to search.

Some criterion classes are used for searching for landmarks (for example, `CPosLmCategoryCriteria` is used for searching for landmarks that contain a certain category) and some are used to search for landmark categories (for example, `CPosLmCatNameCriteria` is used to search for landmark categories by specifying a category name).

Searches can be run in incremental mode. `StartLandmarkSearchL` and `StartCategorySearchL` both return a `CPosLmOperation` object, which is used to execute the search. If it is sufficient to run the search synchronously, the client only has to call `CPosLmOperation::ExecuteL`. If it is run incrementally, the client can supervise the progress of the operation. The `CPosLmOperation::NextStep` function in the search operations cannot be executed synchronously using `User::WaitForRequest`. Doing so may cause the operation to hang. `CPosLmOperation::NextStep` must be run using an active object. The client can cancel the search by deleting the `CPosLmOperation` object.

By default, these functions start a new search, but the client can specify that only the previous matches should be searched. This can be used to refine the search when there are many matches.

It is not allowed to search by category item ID in `CPosLmCategoryCriteria` since an item ID is only valid in one landmark database. It is only allowed to specify a named category, a global category, or no category. It is not allowed to search with `CPosLmIdListCriteria` since an item ID is only valid in one landmark database. In both cases the search functions leave with the error code `KErrArgument`.

During the search execution, a read-lock is acquired for each database. Only one search can be performed at a time by the single instance of the class. If a search is already running, the search function leaves with the error code `KErrInUse`.

After search completion, the client can make a second search and specify that only the matches from the previous search shall be considered.

The client can also set a limit on how many search matches should be returned (see `SetMaxNumOfMatches` in Section 2.5.12.5).

The client retrieves the matches from the search by requesting an iterator (see `MatchIteratorL` in Section 2.5.12.6) or by using display data (see `SetDisplayData` in Section 2.5.12.5).

If `CPosLmMultiDbSearch` is used, the client must call the global function `ReleaseLandmarkResources` before terminating in order to release all the used landmark resources; otherwise the client may receive an ALLOC panic.

The `NetworkServices` capability is required for remote databases.

2.5.12.1 Construction

```
static CPosLmMultiDbSearch* NewL(const CDesCArray&
aDatabaseList)
```

Usage:

A two-phased constructor.

Leaves with `KErrArgument` if the database list is empty.

Parameters:

Parameter	Description
<code>aDatabaseList</code>	An array containing the URIs of the landmark databases to be used in the search.

Returns:

A new instance of this class.

2.5.12.2 Databases to search

```
void SetDatabasesToSearchL(const CDesCArray& aDatabaseList)
```

Usage:

Specifies the list of databases that should be used in the search.

If this function is called and then `StartLandmarkSearchL` or `StartCategorySearchL` is called with the flag `aSearchOnlyPreviousMatches` set, new databases that were not a part of the previous search will generate no matches.

If the client specifies database(s) that do not exist, `GetSearchError` will report the error code `KErrNotFound` for those databases after the search is started.

with `KErrInUse` if called during a search.

If this function is called after a search has completed, database indexes in the results may become invalid.

If a database is removed from a previously set list, the search result of that database is unavailable after this function is called. Search errors are reset after this function is called.

This function will leave with `KErrArgument` if the database list is empty.

Parameters:

Parameter	Description
<code>aDatabaseList</code>	An array containing the URIs of the landmark databases to be used in the search.

```
CDesCArray* DatabasesToSearchL()
```

Usage:

Returns a list of the databases to search. The ownership is transferred to the caller.

Returns:

A list of the databases to search.

2.5.12.3 Starting a search for landmarks

```
CPosLmOperation* StartLandmarkSearchL(
    const CPosLmSearchCriteria& aCriteria,
    TBool aSearchOnlyPreviousMatches = EFalse)
```

Usage:

Starts a landmark search.

If there are no previous matches and the client specifies that previous matches should be used, this function leaves with the error code `KErrArgument`.

If a search is already running, this function leaves with the error code `KErrInUse`.

If the search criterion is not valid for landmark searching, this function leaves with the error code `KErrArgument`.

If the search criterion is not supported, this function leaves with the error code `KErrNotSupported`.

If `CPosLmIdListCriteria` or `CPosLmCategoryCriteria` with an item ID set is used, this function leaves with the error code `KErrArgument`.

The client takes ownership of the returned operation object.

This function requires the `ReadUserData` capability.

Parameters:

Parameter	Description
<code>aCriteria</code>	The search criterion.
<code>aSearchOnlyPreviousMatches</code>	This flag may be used to perform a search within the results of previous search.

Returns:

A handle to the search operation.

```
CPosLmOperation* StartLandmarkSearchL(
    const CPosLmSearchCriteria& aCriteria,
    const TPosLmSortPref& aSortPref,
    TBool aSearchOnlyPreviousMatches = EFalse)
```

Usage:

Starts a landmark search.

This overload of the `StartLandmarkSearchL` function lets the client define the sort order for the search matches.

Only sorting by landmark name is supported. If the client tries to sort by another attribute, this function leaves with the error code `KErrNotSupported`.

If there are no previous matches and the client specifies that previous matches should be used, this function leaves with the error code `KErrArgument`.

If a search is already running, this function leaves with the error code `KErrInUse`.

If the search criterion is not valid for landmark searching, this function leaves with the error code `KErrArgument`.

If the search criterion is not supported, this function leaves with the error code `KErrNotSupported`.

If `CPosLmIdListCriteria` or `CPosLmCategoryCriteria` with an item ID set is used, this function leaves with the error code `KErrArgument`.

The client takes ownership of the returned operation object.

This function requires the `ReadUserData` capability.

Parameters:

Parameter	Description
<code>aCriteria</code>	The search criterion.
<code>aSortPref</code>	A sort preference object.
<code>aSearchOnlyPreviousMatches</code>	This flag may be used to perform a search within the results of previous search.

Returns:

A handle to the search operation.

2.5.12.4 Starting a search for categories

```
CPosLmOperation* StartCategorySearchL(
    const CPosLmSearchCriteria& aCriteria,
    CPosLmCategoryManager::TCategorySortPref aSortPref,
    TBool aSearchOnlyPreviousMatches = EFalse)
```

Usage:

Starts a search for landmark categories.

If there are no previous matches and the client specifies that previous matches should be used, this function leaves with the error code `KErrArgument`.

The criterion, which defines if a landmark category is a match, is passed as input to this function.

If a search is already running, this function leaves with the error code `KErrInUse`.

If the search criterion is not valid for landmark category searching, this function leaves with the error code `KErrArgument`.

If the search criterion is not supported, this function leaves with the error code `KErrNotSupported`.

The client takes ownership of the returned operation object.

This function requires the `ReadUserData` capability.

Parameters:

Parameter	Description
aCriteria	The search criteria.
aSortPref	Sort preference for the search results.
aSearchOnlyPreviousMatches	This flag may be used to perform a search within the results of previous search.

Returns:

A handle to the search operation.

2.5.12.5 Options

```
TInt MaxNumOfMatches() const
```

Usage:

Retrieves the maximum number of search matches limit for each database.

By default, the maximum number of matches is unlimited.

Returns:

The maximum number of search matches for each landmark database involved in the search or `KPosLmMaxNumOfMatchesUnlimited` if the number of matches is unlimited.

```
void SetMaxNumOfMatches(TInt aMaxNumOfMatches =
KPosLmMaxNumOfMatchesUnlimited)
```

Usage:

Sets the maximum number of search matches limit for each database.

This function is used to limit the number of matches retrieved from each database in the search operation. If the limit is set, the search operation will stop when this limit is reached. By default, the maximum number of matches is unlimited.

If a new value for the maximum number of matches is set when a search is ongoing, it will not affect the current search. The new maximum will be utilized in the next search.

Parameters:

Parameter	Description
aMaxNumOfMatches	The maximum number of search matches for each landmark database involved in the search.

```
void SetDisplayData(CPosLmDisplayData& aData)
```

Usage:

Display data can be used as an alternative way to get results from a database search. Landmarks or categories are added to the display data collection during a search depending on the search type.

This function may replace the combination of using `MatchIteratorL` and reading landmark or category data. Result data is read already during the search and no duplicate access to the database is needed.

The display data object will be reset each time a new search is started. No items during the search are removed from the collection. New found matches can be added every time the next search step is completed, see `CPosLmDisplayData::NewItemIndex` in Section 2.5.10.3.

If the client sets display data during an ongoing search, this function panics with the code `EPosSearchOperationInUse`.

The client owns the display data object. If the client deletes it during a search, this may lead to unexpected errors. The client must call `UnsetDisplayData` before it deletes the display data object.

Search results from all databases are collected in the same displayable data collection. `CPosLmDisplayItem::DatabaseIndex` may be used to know to which database every displayable item belongs. The database index matches the databases specified in this object, see `DatabaseUriPtr` in Section 2.5.12.6.

Parameters:

Parameter	Description
aData	The displayable data.

```
void UnsetDisplayData()
```

Usage:

Unsets display data. No further data will be added to the display data set with `SetDisplayData`.

If the client unsets display data during an ongoing search, this function panics with the code `EPosSearchOperationInUse`.

2.5.12.6 Retrieving results

```
TUint TotalNumOfMatches() const
```

Usage:

Returns the total number of matches so far in the search.

This function can also be called during a search operation.

Returns:

The number of search matches.

```
TUint NumOfDatabasesToSearch() const
```

Usage:

Returns the number of databases involved in the search.

Returns:

The number of databases involved in the search.

```
TUint NumOfMatches(TUint aDatabaseIndex) const
```

Usage:

Returns the number of matches so far in the search for a database specified by index.

The index must be strictly less than `NumOfDatabasesToSearch`; otherwise this function will panic with the code `EPosInvalidIndex`. The URI of the database can be retrieved by calling `DatabaseUriPtr`.

This function can also be called during a search operation.

Parameters:

Parameter	Description
<code>aDatabaseIndex</code>	The index of the database for which to get the number of search matches.

Returns:

The number of search matches.

```
TPtrC DatabaseUriPtr(TUint aDatabaseIndex) const
```

Usage:

Returns the URI of a database involved in the search.

The client specifies an index of the database URI to retrieve. The index must be strictly less than `NumOfDatabasesToSearch`; otherwise this function will panic with the code `EPosInvalidIndex`.

Parameters:

Parameter	Description
<code>aDatabaseIndex</code>	The index of the database URI to retrieve.

Returns:

A pointer to the database URI descriptor. This pointer is only valid until `SetDatabasesToSearch` is called, or the `CPosLmMultiDbSearch` object is destroyed, whichever happens first.

```
CPosLmItemIterator* MatchIteratorL(TUint aDatabaseIndex)
```

Usage:

Creates an iterator object to iterate the matching landmarks or categories from one of the databases involved in the search.

The database for which to get the search matches is specified by index. The index must be strictly less than `NumOfDatabasesToSearch`; otherwise this function will panic with the code `EPosInvalidIndex`. The URI of the database can be retrieved by calling `DatabaseUriPtr`.

This function can also be called during a search in order to read the matches encountered so far. Note that the iterator will not iterate any new matches. If new matches are found, a new iterator needs to be created to retrieve them. The previous matches will also be included.

If the client has started a search but no matches have been found yet in the database, an empty iterator is returned.

If a sort preference was specified when the search was started, the landmarks or categories will be sorted when the search is complete but the items are not necessarily sorted if this function is called during a search.

The client takes ownership of the returned iterator object.



Note: The iterator iterates matches in `CPosLmMultiDbSearch`. It cannot be used after the search object has been deleted.

Parameters:

Parameter	Description
<code>aDatabaseIndex</code>	The index of the database for which to get the search matches.

Returns:

A search results iterator.

2.5.12.7 Search errors

```
struct TSearchError
{
    TUint iDatabaseIndex
    TInt iErrorCode
};
```

Usage:

A struct containing a search error.

Public attributes:

Parameter	Description
<code>iDatabaseIndex</code>	The index of the database where the search has failed. The database URI can be retrieved by calling <code>DatabaseUriPtr</code> .
<code>iErrorCode</code>	The search error code for the database.

```
TUint NumOfSearchErrors() const
```

Usage:

Returns the number of errors encountered during the search. This is the same as the number of the databases in which the search has failed.

Returns:

The number of errors encountered during the search.

```
void GetSearchError(TUint aErrorIndex, TSearchError&
aSearchError) const
```

Usage:

Returns the errors encountered in the search.

The client specifies an index of the error to retrieve. The index must be strictly less than `NumOfSearchErrors`; otherwise this function panics with the code `EPosInvalidIndex`. Whenever a new search is started, this function must be called again to retrieve error codes, if any.

Parameters:

Parameter	Description
<code>aErrorIndex</code>	The index of the error to retrieve.
<code>aSearchError</code>	On return, contains the search error.

2.6 Code architecture

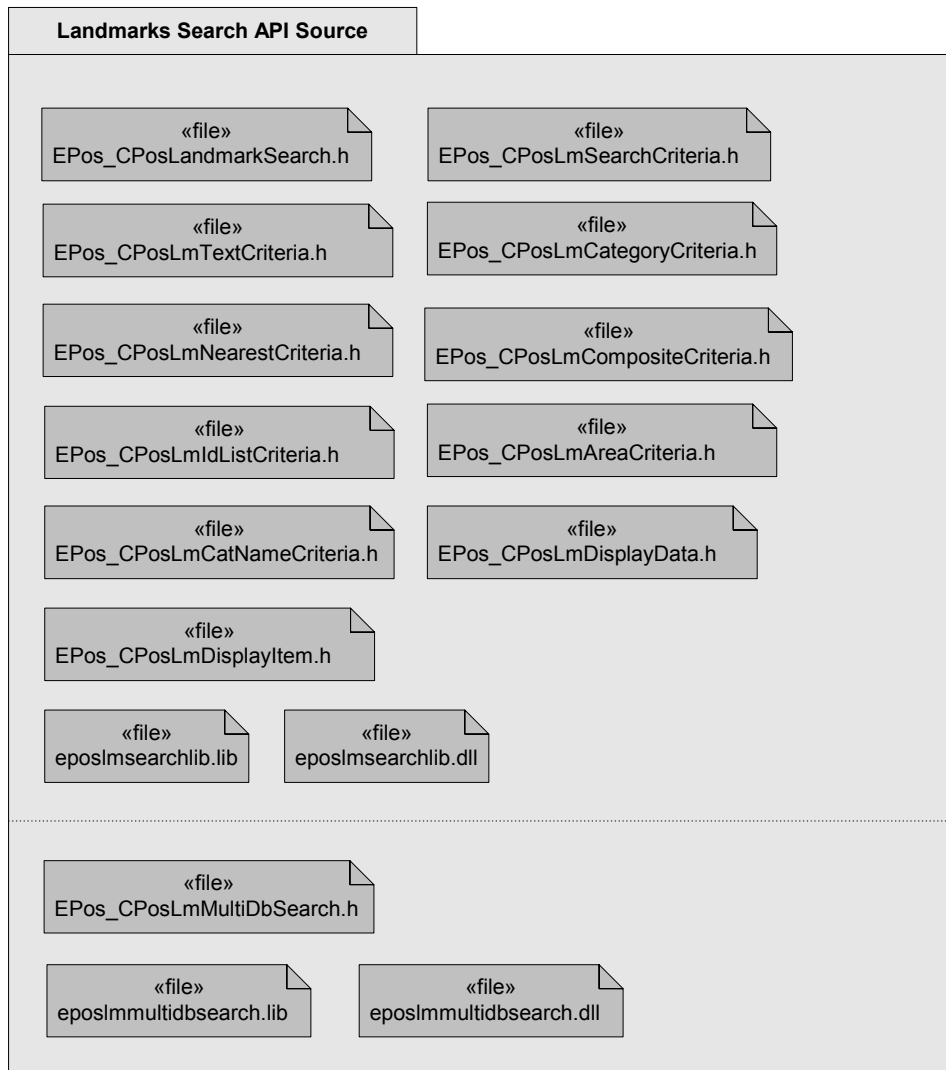


Figure 2.5: Files needed by the Landmarks Search API client

The Landmarks Search API classes are defined in the header files shown in Figure 2.5. `eposlmsearchlib.LIB` contains link-time information for the base part of the API and `eposlmsearchlib.DLL` contains the run-time information. `eposlmmultidbsearch.LIB` contains link-time information for the multiple database part of the API and `eposlmmultidbsearch.DLL` contains the run-time information.

3. Terms and abbreviations

Term or abbreviation	Meaning
API	Application programming interface
DBMS	Database management system, a commonly used term for the Symbian OS database engine.
ECom	Symbian OS framework for plug-in DLLs.
Landmark	A landmark is a named object that contains a location. The location can be defined by various attributes, for example WGS84 coordinates or a textual address.
Landmark attribute	An attribute of a landmark, for example landmark name, position, landmark description, coverage area. For more details, see reference document [2].
Landmark category	A landmark can be categorized by assigning a landmark category to it. A typical landmark category is "Restaurant".
Landmark database	Persistent storage of a collection of landmarks and landmark categories.
Position fields	Generic position fields defined in the Location Acquisition API (see reference document [1]), for example street name, country, building name.
UML	Unified modeling language

4. References

- [1] [S60 Platform: Location Acquisition API Specification](#), available at www.forum.nokia.com
- [2] [S60 Platform: Landmarks API Specification](#), available at www.forum.nokia.com
- [3] Plug-in Architecture 6.2 – ECom Architecture Overview, <http://www.symbian.com/>

5. Evaluate this resource

Please spare a moment to help us improve documentation quality and recognize the resources you find most valuable, by [rating this resource](#).