VARIAN medical systems

Eclipse Algorithm API Reference Guide

Algorithms for Research Purposes

CE 0086 ISO 13485 REGISTERED

P1014090-001-A

JUNE 2015

Document ID	P1014090-001-A, Revision 001-A
Abstract	*Eclipse Algorithm API Reference Guide* provides information about using Eclipse Algorithm API version 13.7. This publication is the English-language original.
Manufacturer	Varian Medical Systems, Inc. 3100 Hansen Way Palo Alto, CA 94304-1038 United States
European Authorized Representative	Varian Medical Systems UK Ltd. Oncology House Gatwick Road, Crawley West Sussex RH10 9RG United Kingdom
Notice	Information in this user guide is subject to change without notice and does not represent a commitment on the part of Varian. Varian is not liable for errors contained in this user guide or for incidental or consequential damages in connection with furnishing or use of this material. This document contains proprietary information protected by copyright. No part of this document may be reproduced, translated, or transmitted without the express written permission of Varian Medical Systems, Inc.
FDA 21 CFR 820 Quality System Regulation (CGMPs)	Varian Medical Systems, Oncology Systems products are designed and manufactured in accordance with the requirements specified within this federal regulation.
ISO 13485	Varian Medical Systems, Oncology Systems products are designed and manufactured in accordance with the requirements specified within ISO 13485 quality standards.
CE	Varian Medical Systems, Oncology Systems products meet the requirements of Council Directive MDD 93/42/EEC.
IEC62083	Eclipse™ Treatment Planning System is IEC62083:2009 compliant.
EU REACH SVHC Disclosure	The link to the current EU REACH SVHC disclosure statement is http://www.varian.com/us/corporate/legal/reach.html.
HIPAA	Varian's products and services are specifically designed to include features that help our customers comply with the Health Insurance Portability and Accountability Act of 1996 (HIPAA). The software application uses a secure login process, requiring a user name and password that supports role-based access. Users are assigned to groups, each with certain access rights, which may include the ability to edit and add data or may limit access to data. When a user adds or modifies data within the database, a record is made that includes which data were changed, the user ID, and the date and time the changes were made. This establishes an audit trail that can be examined by authorized system administrators.
WHO	ICD-0 codes and terms used by permission of WHO, from: ■ International Classification of Diseases for Oncology, (ICD-0) 3rd edition, Geneva, World Health Organization, 2000. ICD-10 codes and terms used by permission of WHO, from: ■ International Statistical Classification of Diseases and Related Health Problems, Tenth Revision (ICD-10). Vols 1-3, Geneva, World Health Organization, 1992.

Contents

Chapter 1 Introduction

The Eclipse Treatment Planning System (Eclipse TPS) is used to plan radiotherapy treatments for patients with malignant or benign diseases. Eclipse TPS is used to plan external beam irradiation with photon, electron and proton beams, as well as for internal irradiation (brachytherapy) treatments. In addition, the Eclipse Proton Eye algorithm is specifically indicated for planning proton treatment of neoplasms of the eye.

Eclipse Algorithm Application Programming Interface (Eclipse Algorithm API, or EAAPI) is a programming interface and a software library for Eclipse that allows algorithm developers to write advanced custom algorithms for many of the calculation types supported by Eclipse and the Distributed Calculation Framework (DCF). These include volume dose calculation and dose-volume optimization algorithms. Algorithms written with EAAPI seamlessly integrate to Eclipse and to DCF, and can be selected for use by the same means as Varian algorithms. The custom algorithms can also be used in conjunction with Varian algorithms.

Note *This feature is for research purposes only. Research must follow national requirements for conducting research and/or clinical studies.*

Who Should Read This Manual

This guide is written mainly for medical/technical personnel who wish to write custom algorithms to be used in Eclipse and the Distributed Calculation Framework (DCF) in a research database.

Note *The system prevents the use of EAAPI with a database that is not defined as a research database.*

It is assumed that you are familiar with:

- Eclipse Treatment Planning System, including the Beam Configuration and Distributed Calculation Framework
- Radiation oncology domain and concepts
- DICOM
- Software engineering practices
- Microsoft Visual Studio development environment
- C++ programming language and object oriented development
- English

Note *Before creating your own algorithms, familiarize yourself with the Eclipse user documentation, especially any safety-related information, cautions and warnings found throughout the documentation.The system prevents the use of EAAPI with a database that is not defined as a research database.*

Visual Cues

This publication uses the following visual cues to help you find information:

⚠	**WARNING**	**A warning describes actions or conditions that can result in serious injury or death.**
⚠	**CAUTION**	**A caution describes hazardous actions or conditions that can result in minor or moderate injury.**
⚠	**NOTICE**	**A notice describes a practice not related to physical injury to include equipment, data, and other issues.**
	Note	*A note describes information that may pertain to only some conditions, readers, or sites.*
	Tip	*A tip describes useful but optional information such as a shortcut, reminder, or suggestion, to help get optimal performance from the equipment or software.*

Related Publications

▪ RT Administration Reference Guide	P1012330-001-A
▪ Beam Configuration Reference Guide	P1012331-001-A
▪ Eclipse Photon and Electron Algorithms Reference Guide	P1012701-001-A
▪ Eclipse Proton Algorithms Reference Guide	P1012702-001-A
▪ Eclipse Scripting API Reference Guide	P1013132-001-A
▪ Eclipse Scripting API Online Help	P1013133-001-A

Contact Varian Customer Support

Varian Customer Support is available on the internet, by e-mail, and by telephone. Support services are available without charge during the initial warranty period.

The MyVarian website provides contact information, product documentation, and other resources for all Varian products.

1. Go to www.MyVarian.com.
2. Choose an option:
3. Click Contact Us at the top of the window to display customer support and training options, and international e-mail addresses and telephone numbers.
   - If you have an account, enter your User login information (email and password).
   - If you do not have an account, click **Create New Account** and follow the instructions. Establishing an account may take up to two working days.
4. From the **Contact Us** page, choose an option:
   - Call Varian Medical Systems support using a phone support number for your geographic area.
   - Complete the form corresponding to your request for use on a call with a live Varian representative; then follow the instructions to complete the remote connect options, and click **Submit**.

   You can order documents by phone, request product or applications support, and report product-related issues. Links on the MyVarian website navigate to other support resources for products, services, and education.
5. To find documents, click Product Documentation.

   Online documents in PDF format include customer technical bulletins (CTBs), manuals, and customer release notes (CRNs).

Chapter 2 About Eclipse Algorithm API

Eclipse Algorithm API (EAAPI) is an object-oriented C++-based programming interface and a software library for Eclipse. EAAPI allows algorithm developers to write advanced custom algorithms for many of the calculation types supported by Eclipse and the Distributed Calculation Framework (DCF) in a dedicated, non-clinical environment. These include volume dose calculation and dose-volume optimization algorithms. Algorithms written with EAAPI seamlessly integrate to Eclipse and to DCF, and are selectable for use by the same means as Varian algorithms. The custom algorithms can also be used in conjunction with Varian algorithms.

WARNING **The authors of custom algorithms are responsible for verifying the accuracy and correctness of the algorithms.**

Features

- Support for writing custom algorithms and integrating them seamlessly into Eclipse/DCF in the same way as Varian algorithms.
- Automatic support for distributed parallel calculation through the DCF.
- Support for the most of the calculation types of Eclipse/DCF:
  - Electron field volume dose calculation
  - Photon field volume dose calculation
  - Photon plan volume dose calculation
  - Photon dose-volume optimization
  - Photon LMC (Leaf Motion Calculation) CBSF (Collimator Back Scatter Factor) calculation
  - Photon LMC monitor units calculation
  - Photon MLC leaf motions calculation
  - Photon actual fluence calculation for IMRT field
  - Proton field volume dose calculation
  - Proton dose-volume optimization
  - Proton ideal compensator calculation
  - Proton milling pattern compensator calculation.
- Support for accessing and utilizing the dose deposition coefficient matrix calculated by the Varian Proton Convolution Superposition algorithm.
- Support for algorithm-specific calculation options through XML definitions.
- Support for custom progress messages, information messages, warning messages and error messages during calculation.
- Support for calculation-specific resource usage estimates for efficient, dynamic scheduling of the calculations.

- Support for binding the algorithm to execute on a specific host (useful if the algorithm is, for instance, designed for specific hardware, like GPU).
- Object-oriented C++ interface to allow wide range of implementation and integration possibilities.
- Example algorithms for all the supported calculation types.
- Utility methods for common tasks.
- User access control to restrict the use of custom algorithms for particular user groups.
- Full 64-bit support.

System Requirements

The basic system requirements of the Eclipse Algorithm API are the same as those of Eclipse. For more information, refer to Eclipse Customer Release Note and Eclipse Distributed Calculation Framework Customer Release Note.

To create and use EAAPI 13.0 algorithms, you need:

- Eclipse version 13.0 or newer.
- DCF version 13.0 or newer.
- Microsoft Visual Studio 2013 *(version "12.0.31101.00 Update 4")* development environment installed on the workstation in which you intend to work with algorithm development. Confirm your version of Microsoft Visual Studio 2013 in Microsoft Visual Studio (choose **Help > About Microsoft Visual Studio**)*. M*ake also sure that your Microsoft Visual Studio 2013 installation has the *X64 Compilers and Tools* option installed to be able to compile for X64 target. *Eclipse Algorithm API is provided with 64-bit (X64) libraries only.*
- License for Eclipse Algorithm API and for any Varian algorithm and Eclipse feature you intend to use.
- Database configured for research use.

Note *Visual Studio is not needed on workstations where the algorithms are executed. It is only needed on workstations where the coding is done. It is recommended not to do the coding on an Eclipse workstation. The customer is responsible for providing the workstations and Visual Studio 2013.*

Version Compatibility

EAAPI 13.0

EAAPI 13.0 is compatible with Eclipse versions 13.0 and newer. Algorithms created with older versions must be recompiled before they can be used with Eclipse 13.0 or newer version.

EAAPI 11.0

EAAPI 11.0 is compatible with Eclipse 11.0.

EAAPI 10.0

EAAPI 10.0 is compatible with Eclipse 10.0.

API versions former to 10.0 are not compatible with Eclipse 10.0 or higher. Since the data objects in the former API versions and in the Eclipse Algorithm API are mostly the same, migrating algorithms from former API versions to the Eclipse Algorithm API should be a relatively straightforward task, assuming that the code and components of the custom algorithm can compile or be utilized from the 64-bit servant.

What's New in EAAPI 13.0

- `ProtonOptimizationCapability` now supports optimization for line scanning (in addition to the optimization for spot weights and spot scanning).
- `ProtonOptimizationCapability` has the following new requests: `GetExpandedTargetSegment`, `GetFieldNumberAndWeightList`, `GetUniqueFractionation`.
- Unicode support. The following methods now expect or return UTF-8 or UTF-16LE encoded strings (as `std::string` or `std::wstring`, respectively):

  a. In `EAAPI::Servant`:
     `SetName`, `SetDescription`, `SetSupportedMachineModels`, `SetSupportedMLCModels`

  b. In capability-specific requesters:
     `GetBeamDataRootDirectoryPath`

  c. `EAAPI::BeamDataUtils::CreateBeamDataAccessor`

  d. In `EAAPI::IBeamDataAccessor`:
     `GetBeamDataRootDirectoryPath`, `GetSpecificBeamDataDirectoryPath`, `PutBeamDataDirectoryInformation`

  e. In capability-specific requesters and in `EAAPI::MessageHelper`:
     `PutErrorFlag`, `PutWarning`, `PutInformation`, `PutProgress`, `PutDebug`

  f. `EAAPI::ISessionProperties::SetAbortMessage`

  The signature of the `EAAPI::Servant::Main` method was changed to a wide-character version.

  `EAAPI::RequestException` has the following new methods: `what_utf8()`, `what_utf16LE()` (the `what()` method has been removed).

  `EAAPI::Utils` class has the following new methods: `ConvertUtf8ToUtf16LE`, `ConvertUtf16LEToUtf8`, `ConvertUtf16LEToACP`.

  `EAAPI::Utils::SaveParserHelperObjectToFile` was renamed to `EAAPI::Utils::SaveParserHelperObjectToFileW`. It now takes the `filePath` argument as a UTF-16LE-encoded string, and returns false on error (instead of raising an exception).
- `EAAPI::IBeamDataAccessor::GetSpecificBeamDataDirectoryPath` no longer raises `EAAPI::BeamDataException` on error. Instead, it returns false on error. The `EAAPI::BeamDataException` class has been removed from the API.

- A bug in an example algorithm, where missing X-jaw information was not handled correctly, has been fixed. The missing information is now read from the MLC positions.
- A bug in proton optimization example algorithm, where a movable distal absorber resulted in an invalid dose, has been fixed.

Eclipse Algorithm API and Distributed Calculation Framework

For information about Distributed Calculation Framework (DCF), refer to *Beam Configuration Reference Guide*.

Algorithms written with Eclipse Algorithm API automatically support distributed parallel calculation, as applicable for the calculation. Based on the calculation type to be performed, Eclipse divides the calculation into sub-tasks and efficiently distributes and parallelizes the execution of the sub-tasks. Usually the distribution and parallelization occurs by field, but can also occur within the field, for instance, for each control point. The example below demonstrates how the distribution and parallelization works.

Algorithms written with EAAPI are DCF servant executables. DCF servant executables, along with any related resource and runtime library files, are copied to the DCF directory and installed to the DCF configuration, after which Eclipse can use them in calculations.

Capability-specific resource estimate expressions affect the estimated resource usage ratio of the calculation nodes and thus the dynamic scheduling.

Example of Distribution and Parallelization

Scenario: Photon Volume Dose calculation is started for the current plan in Eclipse. Plan Dose Calculation option is `OFF`, Control point field parallelization factor is 1, there are neither fluences nor leaf motions in the plan, and `HybridServiceConnector` is in use.

Flow of events:

1. Eclipse divides the calculation into as many sub-tasks as there are fields.
2. Eclipse requests calculation service for each sub-task, often in parallel, until all the sub-tasks have been calculated. The calculation services are provided by the DCF servant executables (Varian or custom). Because `HybridServiceConnector` is in use, Eclipse prefers local calculations over remote calculations when possible.
    a. In the case of local calculation, Eclipse executes the corresponding servant executable in the context of the workstation that Eclipse resides in. Data is transferred directly between Eclipse and the servant, without network intervention. When the corresponding calculation is over, the servant executable exits.

b. Whenever Eclipse requests a remote calculation, it does it through the Distributor[1]. Each Agent[2] in the calculation network is connected to the Distributor, and the Agent announces to the Distributor what calculation services the Agent can provide. The Eclipse calculation request is transferred by the Distributor to an available Agent. The Agent executes the corresponding servant executable in the context of the workstation that the Agent resides in. Data is transferred between Eclipse and the Agent through the Distributor in the network. When the corresponding calculation is over, the servant executable exits.

3. Eclipse collects the results of the sub-tasks and combines them into the final results.

In addition to the parallelization provided by Eclipse/DCF, you can implement further parallelization (for instance, multithreading within the capability) for custom algorithms.

Eclipse Algorithm API and Custom Beam Data

EAAPI supports custom beam data. The support for custom beam data, however, is limited only to the following:

- In Beam Configuration, it is possible to create a calculation model for a custom algorithm. It is also possible to create a mapping from a specific treatment machine / energy mode pair (electrons and photons) or from a specific treatment machine / treatment technique pair (protons) to the corresponding beam data directory path.
  For the mechanism that Beam Configuration uses to manage the mapping, refer to *Beam Configuration Reference Guide*.
- The algorithm resolves a specific treatment machine / energy mode or treatment machine / treatment technique pair with EAAPI to the corresponding beam data directory path configured in Beam Configuration.

NOTICE **The author of custom algorithms is responsible for implementing, testing and documenting the mechanisms, tools and processes related to the custom beam data, including beam data configuration, management, approval, approval checking, validity verification, integrity verification, multi-user access handling and error handling.**

Custom systems or implementations may not override or interfere with the mechanism that Beam Configuration uses to manage the mapping.

Note *When creating a calculation model for a custom algorithm in Beam Configuration, it is only possible to create a new beam data root directory or copy existing beam data from a calculation model of a custom algorithm. Assigning or copying beam data from a calculation model created for a Varian algorithm to a calculation model created for a custom algorithm is not supported.*

[1] The central part of DCF that connects the workstation requesting calculation to the workstation that can provide the requested calculation. Distributor is like a "call center" for distributed calculation purposes.

[2] A Windows service installed and running on each workstation that provides calculation services for other workstations in the network.
Agent launches the algorithm in the workstation it resides in to perform the calculation.

NOTICE **Custom algorithms must prevent dose calculation if the required input data (such as accessory, treatment machine, technique or energy) is unavailable or not supported.**

Installing Eclipse Algorithm API

Before installing Eclipse Algorithm API, make sure that a compatible Eclipse treatment planning system and Distributed Calculation Framework server is installed and properly configured. For more information, see Section "System Requirements" on page 13.

EAAPI is provided in a compressed (zip) archive. The installation is started by extracting the contents of the zip file to a location of your choice, for instance, to a new directory such as `C:\src`. This guide assumes that the installation directory is `C:\src`, but you can install API to any directory you want. It is recommended that you store the extracted EAAPI package as well as any source files and related assets in a version control system.

After extracting the EAAPI package, the top-level directory structure should look as in the following table:

Table 1 Structure of EAAPI Directories

Directory	Description of the contents
`EAAPI`	The base directory of EAAPI library. It is recommended not to change any of the files under this directory.
`BeamDataTemplate`	Basic beam data template files for the custom algorithms.
`bin`	Dynamic link libraries (DLLs) needed at servant run-time. The `x64_Debug` directory contains the DLLs for Debug configuration and `x64_Release` for Release configuration.
`examples`	
`prebuilt`	Prebuilt binaries of the example algorithms, in a directory structure that resembles that of DCF directory.
`source`	Source code for the example algorithms.
`include`	Include (header) files. Add this directory to Additional Include Directories for each of your Visual Studio project that uses EAAPI.
`lib`	Static link libraries needed at algorithm compile time. The `x64_Debug` directory contains the library for Debug configuration and `x64_Release` contains the library for Release configuration.

To complete the installation, copy the beam data template directory from the BeamDataTemplate directory to the templates directory on the DCF server:

1. Copy directory `EAAPI\BeamDataTemplate\__DCF__\client\Templates\[API] (x.y.z)`.
2. Paste the copied directory to `<DCFDirectory>\client\Templates\`.

Note *There is no specific uninstall program for the EAAPI package, because the package contents are intended to be stored in a version control system. Use Windows Explorer to delete the extracted EAAPI files from your system.* Any custom algorithm created with EAAPI that has been installed to the DCF, however, must be uninstalled separately.

Eclipse Algorithm API Concepts

Servant and Executable

Servant refers to the application executable that can provide various calculation services for Eclipse. Servants are standard console-mode applications, created with Visual Studio and linked against the EAAPI libraries (that is why at the end the API code resides inside the servant and not as a separate entity "between" Eclipse and the servant). Linking against EAAPI allows the servant executable to communicate and be compatible with Eclipse and DCF. A servant contains one or more calculation capabilities (of different type). The capabilities all share the same algorithm properties and calculation options. Before a servant can be used in calculations, it needs to be installed in the DCF configuration. Figure 1 is a conceptual representation of a servant in Eclipse Algorithm API.

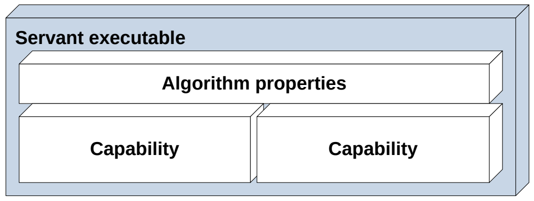


Figure 1 Servant in EAAPI

A servant that implements `PhotonFieldVolumeDoseCapability`, `PhotonFluenceCapability`, `PhotonLMCMonitorUnitsCapability` and `PhotonLMCCBSFCapability` could look like this:

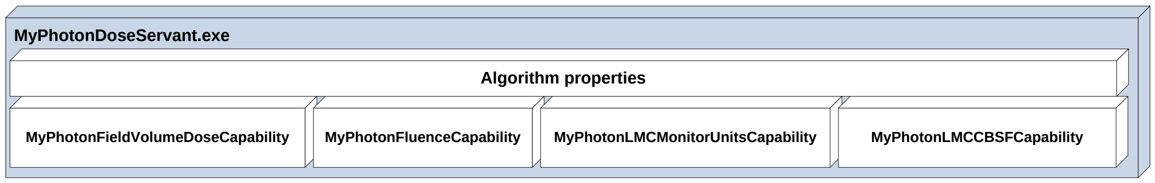

MyPhotonDoseServant.exe
Algorithm properties
MyPhotonFieldVolumeDoseCapability
MyPhotonFluenceCapability
MyPhotonLMCMonitorUnitsCapability
MyPhotonLMCCBSFCapability

Figure 2 Example of a Servant in EAAPI

In brief, a servant is a container that combines the desired calculation capabilities into an entity that can be deployed and utilized.

Algorithm

The custom algorithm code resides inside a capability or capabilities, and the capabilities reside inside a servant. In the context of EAAPI, an algorithm can usually be taken as a synonym for a servant.

Algorithm Property

Algorithm properties specify for Eclipse and the DCF what the algorithm is designed to support. Eclipse and the DCF use the algorithm properties through the DCF configuration to identify and get information about the algorithm. That information is used to manage the calculations as appropriate for the calculation. That is why it is important that your algorithm properties are in line with the implementation of your algorithm.

You define the algorithm properties using `EAAPI::Servant::Set<PropertyName>` methods according to what applies to your algorithm. For instance, to set the algorithm name and version you could issue:

```
EAAPI::Servant::SetName("My Algorithm");
EAAPI::Servant::SetVersion(0, 0, 1, 0);
```

Or, if your algorithm can calculate multiple control points for dynamic arcs, you would issue:

```
EAAPI::Servant::SetCanCalculateMultipleControlPoints(true);
```

For details about different algorithms properties, see Chapter 10 "Algorithm Property Reference" on page 91, the default values and which algorithm properties apply to which capabilities.

Capability

Eclipse contains different calculation types, such as (Photon) Volume Dose, (Photon) Optimization and (Proton) Compensator Conversion. In EAAPI, they are referred to as capabilities, which are entities that perform the calculation. A capability in EAAPI summarizes the specifics of the calculation type, such as the input and output. EAAPI has a base-class for each supported capability, and you derive your custom implementations from those base classes. For instance, `EAAPI::IPhotonFieldVolumeDoseCapability` is the base-class for photon field volume-dose calculation, and `EAAPI::IProtonOptimizationCapability` is the base-class for proton volume-dose optimization.

Each servant must contain at least one capability. You can implement and bundle multiple capabilities into a single servant, if for instance, the capabilities logically belong to the same algorithm family. In general, it is recommended not to bundle multiple capabilities into a single servant unless you need to.

Note that a servant cannot contain multiple capabilities of the same type.

For the capabilities supported by Eclipse Algorithm API, see Chapter 8 "Capability Reference" on page 52.

Requester and Request

Capabilities use *requesters* to communicate with Eclipse. When your capability is executing, EAAPI provides it with a reference to the requester object. The capability is able to communicate with Eclipse through the requester object by issuing *requests*. The requests can, for instance, retrieve input data, or send intermediate or final output. The example in Figure 3 could be a very first draft implementation for a custom `DoHandlePhotonPlanVolumeDoseSession` method:

```
void MyPhotonPlanVolumeDoseCapability::DoHandlePhotonPlanVolumeDoseSession(const
EAAPI::ISessionProperties& sessionProperties,
EAAPI::IPhotonPlanVolumeDoseRequester& requester)
{
      std::auto_ptr<DCRVAPlan> planAutoptr = requester.GetPlan110();

      std::vector<EAAPI::FieldNumberAndWeight> fieldNumberAndWeightList;
      requester.GetFieldNumberAndWeightList(fieldNumberAndWeightList);

      std::auto_ptr<DCRVADensity> densityAutoptr = requester.GetDensity();

      std::auto_ptr<DCRVAStructureSet> structureSetAutoptr =
requester.GetStructureSet();

      std::string calculationOptionsXmlString =
requester.GetCalculationOptionsXmlString();

      requester.PutCalculationStartNotification();

      std::auto_ptr<DCRVADose> planDoseAndIMRTActualBeamFluencesAutoptr =
CalculateThePlanDose(planAutoptr.get(), fieldNumberAndWeightList,
densityAutoptr.get(), structureSetAutoptr.get(), calculationOptionsXmlString);

      requester.PutCalculationFinishedNotification();


requester.PutPhotonPlanVolumeDoseFinalOutput110(*planDoseAndIMRTActualBeamFluenc
esAutoptr);
}
```

Figure 3 Example of a Custom Method

First, the code uses the requester object to retrieve the plan, field number and weight list, density image, structure set and the calculation options from Eclipse. Then the code calculates the dose (and notifies Eclipse when the calculation starts and ends). As the last step, the code uses another request to send the final output to Eclipse, after which the method returns.

It is important to note that each request made using a requester can either succeed or fail. A request could fail, for instance, because the requested data is not available in Eclipse for the current plan or in the current Eclipse context. In such cases, EAAPI raises an `EAAPI::RequestException`. Note that the exception is raised on the servant side and that the Eclipse user interface does not show any error. The `EAAPI::RequestException` does not derive from any base-class, so you have to explicitly catch it if you want to respond to it. The `EAAPI::RequestException` class has the following methods that give more details about reason of the failure: `GetErrorCode()`, `what_utf8()` and `what_utf16LE()`:

- The `GetErrorCode()` method is of type `EAAPI::ErrorCode` (an enum), and it can provide additional information about the reason of the request failure in a programmatically recognizable form. For instance,

`EAAPI::ErrorCode_404_NotFound`	indicates that the requested object/data was not found.
`EAAPI::ErrorCode_501_NotImplemented`	indicates that Eclipse does not recognize the request or is not designed to support it in the context the request was made.

  For a list of error codes, refer to `EAAPI::ErrorCode` in `include/EAAPI/RequestException.h`. When designing algorithms, bear in mind that it is possible that more error codes are added to EAAPI in future releases.

- The `what_utf8()` and `what_utf16LE()`methods return a technical error message giving additional details about the reason of the failure. The error message returned by `what_utf8()`and `what_utf16LE()` is not designed to be shown for the end user. Your algorithm should catch the `EAAPI::RequestException` and transform the request failure into a clear error message for the end user, preferably in the local language, depending on what the algorithm attempted to accomplish with the request in the particular situation.

A request can also result in some other exception, such as `std::runtime_error`. Because the reason for such an exception is not as well known as for `EAAPI::RequestException`, the caller should consider it as an error and signal it to Eclipse using the `requester.PutErrorFlag` method and exit the capability instead of trying to recover. `PutErrorFlag`, `PutWarning`, `PutInformation`, `PutProgress` and `PutDebug` are not expected to throw `EAAPI::RequestException`, so your code does not have to prepare for that.

Each capability in EAAPI uses its own capability-specific requester, which contains only those requests that the capability can support. The requester object is valid only while the EAAPI method that passed it to your code is in progress. After the method returns, the object shall not be used anymore (for instance, through a stored reference or from a different thread).

It is recommended to centralize the communication with Eclipse (perhaps excluding progress, information, warning and debug messages) to the capability-specific session handling method (to the callback methods of EAAPI), because they have the reference to the requester object. It is not recommended to pass the requester object around, but instead the actual data objects that were fetched using the requester. The example in Figure 3 illustrates this approach: `CalculateThePlanDose` does not get a requester object as a parameter, but instead it gets pointers to the data objects that were fetched using the requester in the `DoHandlePhotonPlanVolumeDoseSession` method.

Many of the return values from the request methods are `std::auto_ptr`'s of a specific type. `auto_ptr` is a class template in the C++ Standard Library. By definition, an `auto_ptr` is the sole entity responsible for the lifetime of the contained object. So, whenever you get an `auto_ptr` from EAAPI, you know that the `auto_ptr` owns the contained object (which is not owned by, for instance, EAAPI), and you can devise your code accordingly. For instance, you can let the `std::auto_ptr` automatically destroy the contained object when the `auto_ptr` goes out of scope, or you can destruct the contained object whenever appropriate by calling `std::auto_ptr::reset()`, or you can transfer the ownership by calling `std::auto_ptr::release()` (which releases the ownership and returns the contained pointer to the caller to `own/destroy`). Unless otherwise documented (either in this document or in the EAAPI header files), request methods do not return an `auto_ptr` holding a NULL value.

Each request causes EAAPI to communicate with Eclipse. EAAPI does not cache the requests, which means that if you call `requester.GetStructureSet()` twice, the data is also twice transmitted from Eclipse to the servant. Keep this in mind in your implementation, because fetching the same data many times could have noticeable performance implications.

Chapter 3 Creating a Custom Algorithm

Overview

Note

A servant executable should be a console-mode application, designed so that it can be always run in a non-interactive mode. It should never show message boxes or dialogs which could interrupt the algorithm and require user intervention, because the DCF Distributor can schedule the servant to be executed on any suitable workstation.

NOTICE

Custom algorithms must prevent dose calculation if the required input data (such as accessory, treatment machine, technique or energy) is unavailable or not supported.

To create your own algorithm servant, you need to:

- Decide which calculation capabilities you want to implement (for more information, see Chapter 8 "Capability Reference" on page 52).
- Create a Visual Studio project for your servant.
- Define the entry point of the servant (the "main" function in `main.cpp`).
- Define the algorithm name and version, and any additional algorithm properties.
- Implement the capabilities and add them to your servant.
- Add calculation options to your algorithm.

For step-by-step instructions for creating an algorithm servant, see the following sections.

Creating a Visual Studio Project and a Main Function for the Servant

The servant executable that you create must be a console-mode application (`/SUBSYSTEM:CONSOLE`). The file name of the servant executable must end with `Servant.exe` (for instance, `MyServant.exe`). Also, any code that uses EAAPI libraries must be configured as follows:

- Compile to X64 target (`/MACHINE:X64`)
- Use Visual C++ Multi-threaded Runtime Library DLLs (`/MD` for Release or `/MDd` for Debug)
- C++ exceptions enabled (`/EHsc`)
- Struct Member Aligment as the default (`/Zp8`)
- Checked Iterators disabled (`_SECURE_SCL=0`)
- Debug Iterators disabled (`_HAS_ITERATOR_DEBUGGING=0`)

In your Visual Studio projects, use the default Visual Studio naming. This makes it less error-prone to configure the settings for the Linker Additional Dependencies, because you can use Visual Studio macros `$(PlatformName)` and

`$(ConfigurationName)` to specify the EAAPI library file name that is to be linked in. The default naming is the following:

- X64 platform name "`x64`"
- Release configuration name "`Release`"
- Debug configuration name "`Debug`"

To Create the Visual Studio Project (.vxproj) and Visual Studio Solution (.sln) for Your Servant

1. Choose **File > New > Project**.
2. Select project template **Visual C++ > Win32 Console Application**.
   The platform will be changed to X64 in later steps.
3. Select **.NET Framework 4**.
4. Enter a project name for the servant, for instance, "MyServant".
5. Enter the location where the project directory will be created, for instance, "C:\src".
6. Clear the **Create directory for solution** check box.
7. Click **OK**.
8. In the Win32 Application Wizard, click **Next**.
9. Select the **Empty project** check box.
10. Click **Finish**.
    Visual Studio creates a solution and a project file for your servant (`MyServant.sln` and `MyServant.vcxproj`).

To Change the Win32 Platform Configuration to X64

1. From the Visual Studio main menu, choose **Build > Configuration Manager**.
2. From the **Active solution platform** list, select **<New…>**.
3. From the **Type or select the new platform** list, select **x64**.
   If you cannot see x64 in the drop-down list, see Section "System Requirements" on page 13.
4. From the **Copy settings from** list, select **Win32**.
5. Select the **Create new project platforms** check box.
6. Click **OK**.
7. From the **Active solution platform** list, select **<Edit…>**.
8. Select **Win32**.
9. Click **Remove**.
10. To confirm the removal, click **Yes**.
11. To close the Edit Solution Platforms dialog, click **Close**.
12. From the **Project contexts' Platform** list, select **<Edit…>**.
13. Select **Win32**.
14. Click **Remove**.
15. To confirm the removal, click **Yes**.

16. To close the Edit Project Platforms dialog, click **Close**.
17. To close the Configuration Manager dialog, click **Close**.

    The Win32 platform is deleted from the solution and from the project file, and Visual Studio creates an x64 platform to the solution and to the project file.

To Add `main.cpp` to Your Project File

1. From the Solution Explorer, right-click **Source Files** and choose **Add > New Item**.
2. Select template **C++ File (.cpp)**.
3. To the **Name** text box, type "main.cpp".
4. Click **Add**.
5. Add the following code to `main.cpp` (change the algorithm name and description to the applicable ones):

```
#include "EAAPI/Servant.h"

int wmain(int argc, wchar_t* argv[])
{
    // Set the algorithm properties here
    EAAPI::Servant::SetName("My Algorithm");
    EAAPI::Servant::SetVersion(0, 0, 1, 0);
    EAAPI::Servant::SetDescription("My algorithm description");
    // ...

    // Instantiate and add the custom capabilities to the servant here
    // ...

    // Pass control to the EAAPI
    return EAAPI::Servant::Main(argc, argv);
}
```

6. To start changing the project settings of your servant to the required ones, in the Solution Explorer, right-click the servant project and select **Properties**.
7. From the configuration drop-down list, select **Release**.
8. Change the configuration properties to match the properties listed in Table 2 on page 26 and Table 3 on page 27 for the Release configuration. *Before settings related to C/C++ can be configured for the project, the project must contain a C/C++ source code file.*
9. Click **Apply**.
10. From the configuration drop-down list, select **Debug**.
11. Change the configuration properties to match what is listed in Table 2 on page 26 and Table 3 on page 27 for the Debug configuration.
12. Click **OK**.
13. To complete the set-up of the Visual Studio project and the main function of the servant, verify that both the Debug and Release configurations build without errors.

Any project/code that uses EAAPI libraries must use the Visual Studio project settings listed in Table 2 on page 26. In addition, the servant project must be configured to use the settings listed in Table 3 on page 27. Specifically note that Checked Iterators and Debug Iterators must be disabled (`_SECURE_SCL=0` and `_HAS_ITERATOR_DEBUGGING=0`, respectively). Your projects can contain additional Preprocessor Definitions and Linker Dependencies to those listed in the tables. The differences between the Release and the Debug configurations are highlighted in the tables.

Note *Your servant should exit so that the entry-point function of its primary thread (main thread) returns the value that it received from the* `EAAPI::Servant::Main` *method to the C/C++ runtime, as shown in the above code example. This ensures the following:*

- *Resources of the primary thread are properly cleaned up by the C/C++ runtime (for instance, the runtime calls the destructors of auto objects).*
- *Process exits despite other unterminated threads running in the process (the runtime calls* `ExitProcess` *in the end).*
- *Return value (the error code) from* `EAAPI::Servant::Main` *is passed to the DCF.*

Table 2: Visual Studio Project Settings for Project/Code Using EAAPI Libraries

Visual Studio Project Setting	Release Configuration Value	Debug Configuration Value
General > Character Set	`Use Multi-Byte Character Set`	`Use Multi-Byte Character Set`
C/C++ > General > Additional Include Directories	`<PathToEAAPIBaseDirectory>\include`	`<PathToEAAPIBaseDirectory>\include`
C/C++ > Preprocessor > Preprocessor Definitions	`WIN32;NDEBUG;_SECURE_SCL=0;%(PreprocessorDefinitions)`	`WIN32;_DEBUG;_SECURE_SCL=0;_HAS_ITERATOR_DEBUGGING=0;%(PreprocessorDefinitions)`
C/C++ > Code Generation > Enable C++ Exceptions	`Yes (/EHsc)`	`Yes (/EHsc)`
C/C++ > Code Generation > Runtime Library	`Multi-threaded DLL (/MD)`	`Multi-threaded Debug DLL (/MDd)`
C/C++ > Code Generation > Struct Member Aligment	`Default`	`Default`

Visual Studio Project Setting	Release Configuration Value	Debug Configuration Value
Linker > Input > Additional Dependencies	**`rpcrt4.lib;<PathToEAAPIBaseDirectory>\lib\$(PlatformName)_$(ConfigurationName)\EAAPI_$(PlatformName)_$(ConfigurationName).lib;`**`kernel32.lib;user32.lib;gdi32.lib;winspool.lib;comdlg32.lib;advapi32.lib;shell32.lib;ole32.lib;oleaut32.lib;uuid.lib;odbc32.lib;odbccp32.lib;%(AdditionalDependencies)`	**`rpcrt4.lib;<PathToEAAPIBaseDirectory>\lib\$(PlatformName)_$(ConfigurationName)\EAAPI_$(PlatformName)_$(ConfigurationName).lib;`**`kernel32.lib;user32.lib;gdi32.lib;winspool.lib;comdlg32.lib;advapi32.lib;shell32.lib;ole32.lib;oleaut32.lib;uuid.lib;odbc32.lib;odbccp32.lib;%(AdditionalDependencies)`
Linker > Advanced > Target Machine	`MachineX64 (/MACHINE:X64)`	`MachineX64 (/MACHINE:X64)`

Table 3: Additional Visual Studio Project Settings for the Servant (.exe) Project

Visual Studio project setting	Release configuration value	Debug configuration value
General > Configuration Type	`Application (.exe)`	`Application (.exe)`
Linker > System > SubSystem	`CONSOLE (/SUBSYSTEM:CONSOLE)`	`CONSOLE (/SUBSYSTEM:CONSOLE)`
C/C++ > Preprocessor > Preprocessor Definitions	`WIN32;`**`NDEBUG`**`;_CONSOLE;_SECURE_SCL=0;%(PreprocessorDefinitions)`	`WIN32;`**`_DEBUG`**`;_CONSOLE;_SECURE_SCL=0;`**`_HAS_ITERATOR_DEBUGGING=0`**`;%(PreprocessorDefinitions)`

Setting the Algorithm Properties

Each algorithm must have a name and the version defined. Whether other algorithm properties need to be defined, depends on what capabilities your algorithm contains and what functionality your capabilities support. For details about algorithm properties that apply to the capabilities you are implementing in your servant, see Chapter 10 "Algorithm Property Reference" on page 91.

For instance, if your servant contains a `PhotonFieldVolumeDoseCapability` which supports control point parallelization, you define the corresponding algorithm property to `true` in the main function of your servant:

```
    EAAPI::Servant::SetCanCalculateSeparateControlPoints(true);
```

The algorithm properties must be defined in the main function of your servant executable before you call `EAAPI::Servant::Main`. Your algorithm should not modify the algorithm property once it is set.

Creating and Adding the Capabilities

Each servant has to contain at least one capability. For a description of capabilities, see Chapter 8 "Capability Reference" on page 52.

Each custom capability is a C++ class. To create your own capability, derive it from one of EAAPI capability base-classes and implement the abstract (pure virtual) member functions of the corresponding base-class. Different capabilities have slightly different abstract member functions, depending on the nature of the capability. For details, see Chapter 8 "Capability Reference" on page 52 and the header files.

For example, a `PhotonFieldVolumeDoseCapability` class, whose name is "`MyPhotonFieldVolumeDoseCapability`" is created in the following way:

```
#pragma once

#include "EAAPI/IPhotonFieldVolumeDoseCapability.h"

class MyPhotonFieldVolumeDoseCapability : public
EAAPI::IPhotonFieldVolumeDoseCapability
{
public:

    virtual void DoDefineResourceUsageEstimates(EAAPI::IResourceUsageEstimates&
resourceUsageEstimates);

    virtual void DoHandlePhotonFieldVolumeDoseSession(const
EAAPI::ISessionProperties& sessionProperties,
EAAPI::IPhotonFieldVolumeDoseRequester& requester);
};
```

As seen in the example, the abstract (pure virtual) methods of EAAPI capabilities begin with "`Do`" to emphasize that they are EAAPI "callbacks" and to distinguish them from the possible non-EAAPI methods in the same class.

Creating PhotonFieldVolumeDoseCapability

For volume dose calculation, EAAPI provides you with the following base classes from which you derive your capabilities:

```
EAAPI::IElectronFieldVolumeDoseCapability
EAAPI::IPhotonFieldVolumeDoseCapability
EAAPI::IPhotonPlanVolumeDoseCapability
EAAPI::IProtonFieldVolumeDoseCapability
```

The following example creates a `PhotonFieldVolumeDoseCapability`, but the principles and the approach are the same for `ElectronFieldVolumeDoseCapability` and `ProtonFieldVolumeDoseCapability`. For the differences between the capabilities, see Chapter 8 "Capability Reference" on page 52 and the header files.

Note that since a servant that contains `PhotonFieldVolumeDoseCapability` or `PhotonPlanVolumeDoseCapability` must also contain a `PhotonFluenceCapability`, you need to add that capability to the servant, too.

To Add PhotonFieldVolumeDoseCapability and PhotonFluenceCapability to the Algorithm

1. Change `main.cpp` to contain the following code that instantiates and adds `MyPhotonFieldVolumeDoseCapability` to the servant:

```
#include "EAAPI/Servant.h"
#include "MyPhotonFieldVolumeDoseCapability.h"
#include "MyPhotonFluenceCapability.h"

int wmain(int argc, wchar_t* argv[])
{
    // Set the algorithm properties here
    EAAPI::Servant::SetName("My Algorithm");
    EAAPI::Servant::SetVersion(0, 0, 1, 0);
    EAAPI::Servant::SetDescription("My algorithm description");
    EAAPI::Servant::SetSupportedMachineModels("2300C");

    // Instantiate and add the custom capabilities to the algorithm here

    MyPhotonFieldVolumeDoseCapability myPhotonFieldVolumeDoseCapability;
    EAAPI::Servant::AddPhotonFieldVolumeDoseCapability(myPhotonFieldVolume
DoseCapability);

    MyPhotonFluenceCapability myPhotonFluenceCapability;
    EAAPI::Servant::AddPhotonFluenceCapability(myPhotonFluenceCapability);

    // Pass control to the EAAPI
    return EAAPI::Servant::Main(argc, argv);
}
```

2. To implement the capability, add new files `MyPhotonFieldVolumeDoseCapability.h`, `MyPhotonFieldVolumeDoseCapability.cpp`, `MyPhotonFluenceCapability.h` and `MyPhotonFluenceCapability.cpp` to your Visual Studio project.

3. Add the following code to `MyPhotonFieldVolumeDoseCapability.h`:

```
#pragma once

#include "EAAPI/IPhotonFieldVolumeDoseCapability.h"

class MyPhotonFieldVolumeDoseCapability : public
EAAPI::IphotonFieldVolumeDoseCapability
{
public:

    virtual void DoDefineResourceUsageEstimates(EAAPI::IResourceUsageEstimates&
resourceUsageEstimates);

    virtual void DoHandlePhotonFieldVolumeDoseSession(const
EAAPI::ISessionProperties& sessionProperties,
EAAPI::IPhotonFieldVolumeDoseRequester& requester);
};
```

4. Add a reference to the example project `MyUtils` to bring in the example dose calculation code used in the following code sample. To do this, add `EAAPI\examples\source\MyUtils\MyUtils.vcxproj` to your solution and then from `MyServant`'s Property Pages select **Common Properties > Framework and References > Add New Reference** and choose `MyUtils`.

5. Add the following code to file `MyPhotonFieldVolumeDoseCapability.cpp`:

```
#include "MyPhotonFieldVolumeDoseCapability.h"

#include "../examples/source/MyUtils/PhotonCalcUtils.h"

void
MyPhotonFieldVolumeDoseCapability::DoDefineResourceUsageEstimates(EAAPI::IResour
ceUsageEstimates& resourceUsageEstimates)
{
    resourceUsageEstimates.SetExpectedProcessorUsageExpression("1");
    resourceUsageEstimates.SetExpectedMemoryUsageInMBExpression("200");
}

void
MyPhotonFieldVolumeDoseCapability::DoHandlePhotonFieldVolumeDoseSession(const
EAAPI::ISessionProperties& sessionProperties,
EAAPI::IPhotonFieldVolumeDoseRequester& requester)
{
    try {

        // For the example purposes, let's not use beam data
        EAAPI::BeamDataUtils::StateThatCapabilityDoesNotUseBeamData();

        std::auto_ptr<DCRVAPlan> planAutoptr = requester.GetPlan110();

        int fieldNumber;
        EAAPI::DoseMatrixType doseMatrixType;
        bool useControlPointParallelization;
        requester.GetFieldNumberAndParams(fieldNumber, doseMatrixType,
useControlPointParallelization);
        std::auto_ptr<VACalculationArea> calculationVolumeAutoptr =
requester.GetCalculationVolume();

        std::auto_ptr<DCRVADensity> densityAutoptr = requester.GetDensity();

        std::auto_ptr<DCRVAStructureSet> structureSetAutoptr =
requester.GetStructureSet();

        DicomRTBeam* beam = MyUtils::PhotonCalcUtils::GetBeam(planAutoptr.get(),
fieldNumber);
        if (beam == NULL) {
            throw std::runtime_error("Incorrect plan data. Field data is
missing.");
        }
        double jawX1, jawX2, jawY1, jawY2;
        MyUtils::PhotonCalcUtils::ReadJawPositionsForCurrentField(beam, jawX1,
jawX2, jawY1, jawY2);

        requester.PutCalculationStartNotification();

        std::auto_ptr<DCRVADose> doseAutoptr(new DCRVADose);

        MyUtils::PhotonCalcUtils::CalculateTestDose(densityAutoptr.get(),
planAutoptr.get(), calculationVolumeAutoptr.get(), fieldNumber, jawX1, jawX2,
jawY1, jawY2, doseAutoptr.get());

        requester.PutCalculationFinishedNotification();

        requester.PutPhotonFieldVolumeDoseFinalOutput110(fieldNumber,
*doseAutoptr, doseMatrixType);

    } catch (const std::exception& e) {
        requester.PutErrorFlag(e.what());
    }
}
```

6. Add the following code to `MyPhotonFluenceCapability.h`:

```
#pragma once

#include "EAAPI/IPhotonFluenceCapability.h"

class MyPhotonFluenceCapability : public EAAPI::IPhotonFluenceCapability
{
public:

    virtual void DoDefineResourceUsageEstimates(EAAPI::IResourceUsageEstimates&
resourceUsageEstimates);

    virtual void DoHandlePhotonFluenceSession(const EAAPI::ISessionProperties&
sessionProperties, EAAPI::IPhotonFluenceRequester& requester);
};
```

7. Add the following code to file `MyPhotonFluenceCapability.cpp`:

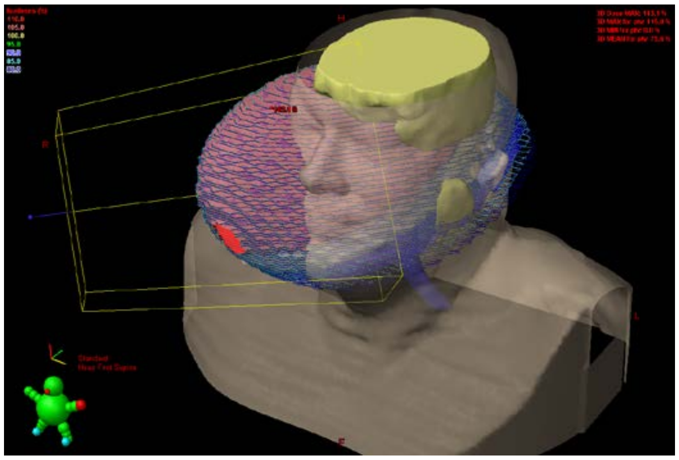
```
#include "MyPhotonFluenceCapability.h"

void
MyPhotonFluenceCapability::DoDefineResourceUsageEstimates(EAAPI::IResourceUsageE
stimates& resourceUsageEstimates)
{
    resourceUsageEstimates.SetExpectedProcessorUsageExpression("1");
    resourceUsageEstimates.SetExpectedMemoryUsageInMBExpression("200");
}

void MyPhotonFluenceCapability::DoHandlePhotonFluenceSession(const
EAAPI::ISessionProperties& sessionProperties, EAAPI:: IPhotonFluenceRequester&
requester)
{
  // Implement this capability if you need support for:
  // - use of PDIP (Portal Dose Image Prediction) algorithm for split fields
  // - visualization of actual fluences of split fields after plan approval
  requester.PutErrorFlag("Not implemented.");
}
```

8. Verify that the solution compiles (both Debug and Release configurations).
9. Create calculation options XML and XSD files for your algorithm (see “Adding Calculation Options to Your Algorithm” on page 41.
10. Install your algorithm to Eclipse/DCF (see "Installing and Uninstalling a Custom Algorithm" on page 44) and calculate with it.

    You should see the resulting test dose:



Congratulations! The dose in Eclipse was calculated by your algorithm.

Creating PhotonOptimizationCapability

For dose-volume optimization, EAAPI provides you the following base classes from which you derive your own capabilities:

`EAAPI::IPhotonOptimizationCapability`
`EAAPI::IProtonOptimizationCapability`

This example creates `PhotonOptimizationCapability`, but the principles and the approach are the same for `ProtonOptimizationCapability`. For the differences between capabilities, see Chapter 8 "Capability Reference" on page 52 and the header files.

To Add PhotonOptimizationCapability to the Algorithm

1. Change `main.cpp` to contain the following code to instantiate and add `MyPhotonOptimizationCapability` to the servant:

```
#include "EAAPI/Servant.h"
#include "MyPhotonOptimizationCapability.h"

int wmain(int argc, wchar_t* argv[])
{
    // Set the algorithm properties here
    EAAPI::Servant::SetName("My Algorithm");
    EAAPI::Servant::SetVersion(0, 0, 1, 0);
    EAAPI::Servant::SetDescription("My algorithm description");
    EAAPI::Servant::SetSupportedMachineModels("2300C");

    // Instantiate and add the custom capabilities to the algorithm here
    MyPhotonOptimizationCapability myPhotonOptimizationCapability;
    EAAPI::Servant::AddPhotonOptimizationCapability(myPhotonOptimization
    Capability);

    // Pass control to the EAAPI
    return EAAPI::Servant::Main(argc, argv);
}
```

2. Add new files `MyPhotonOptimizationCapability.h` and `MyPhotonOptimizationCapability.cpp` to your Visual Studio project.
3. Add the following code to `MyPhotonOptimizationCapability.h`:

```
#pragma once

#include "EAAPI/IPhotonOptimizationCapability.h"

class MyPhotonOptimizationCapability : public
EAAPI::IPhotonOptimizationCapability
{
public:

    virtual void DoDefineResourceUsageEstimates(EAAPI::IResourceUsageEstimates&
resourceUsageEstimates);

    virtual void DoGetInput(const EAAPI::ISessionProperties& sessionProperties,
EAAPI::IPhotonOptimizationRequester& requester);

    virtual void DoProcessOneIteration(
      EAAPI::IPhotonOptimizationRequester& requester,
      std::auto_ptr<DCRVAConstraintModule> newConstraintsAutoptr,
      std::auto_ptr<DCRVAOzParameterModule> newParametersAutoptr,
      DCRVAOzOutput& finalOutput,
      bool& stopOptimizationAndExitServant);

private:

    std::auto_ptr<DCRVADensity> m_densityAutoptr;
};
```

4. Add the following code to `MyPhotonOptimizationCapability.cpp`:

```
#include "MyPhotonOptimizationCapability.h"

void MyPhotonOptimizationCapability:: DoDefineResourceUsageEstimates
(EAAPI::IResourceUsageEstimates& resourceUsageEstimates)
{
    resourceUsageEstimates.SetExpectedProcessorUsageExpression("1");
    resourceUsageEstimates.SetExpectedMemoryUsageInMBExpression("200");
}

void MyPhotonOptimizationCapability::DoGetInput(const EAAPI::ISessionProperties&
sessionProperties, EAAPI::IPhotonOptimizationRequester& requester)
{
    // Fetch your input data here
    m_densityAutoptr = requester.GetDensity();
}

void MyPhotonOptimizationCapability::DoProcessOneIteration(
    EAAPI::IPhotonOptimizationRequester& requester,
    std::auto_ptr<DCRVAConstraintModule> newConstraintsAutoptr,
    std::auto_ptr<DCRVAOzParameterModule> newParametersAutoptr,
    DCRVAOzOutput& finalOutput,
    bool& stopOptimizationAndExitServant)
{
    // Add the code to calculate one iteration.
    // Populate the results of the iteration into finalOutput object.
}
```

5. Implement the `DoGetInput` and `DoProcessOneIteration` methods. For an example, see `EAAPI\examples\source\MyPhotonOptimizationServant`.
6. Verify that the solution compiles (both Debug and Release configurations).
7. To give your servant a go, install it to the DCF and run an optimization with it in Eclipse.

Creating Other Capabilities

Other capabilities are created in the same way: derive your capability from one of the base-classes provided by EAAPI and implement the abstract (pure virtual) member functions of that base-class. For details, see Chapter 8 "Capability Reference" on page 52.

Defining Resource Usage Estimates

The DCF supports dynamic load balancing for efficient resource utilization. Each capability in EAAPI must specify resource usage estimates applicable for the corresponding algorithm. There are two types of resource usage estimates:

- `ExpectedProcessorUsageExpression`, which defines the expected processor usage of the capability.
- `ExpectedMemoryUsageInMBExpression`, which defines the expected memory usage of the capability (in megabytes).

The DCF uses these expressions to determine on which machine to execute the capability.

Each capability in Eclipse Algorithm API is derived from `EAAPI::IResourceUsageEstimable`. To define the estimated resource usage of your algorithm, implement the following abstract method (derived from `EAAPI::IResourceUsageEstimable`) for your capability:

```
virtual void DoDefineResourceUsageEstimates(EAAPI::IResourceUsageEstimates&
resourceUsageEstimates) = 0;
```

The `resourceUsageEstimates` object contains the following methods for your use:

- `SetExpectedProcessorUsageExpression(const std::string& expectedProcessorUsageExpression)`
- `SetExpectedMemoryUsageInMBExpression(const std::string& expectedMemoryUsageInMBExpression)`
- `SetResourceUsageHelperVariableDefinitions(const std::string& resourceUsageHelperVariableDefinitions)`
- `SetHostRequirementsExpression(const std::string& hostRequirementsExpression)`

ExpectedProcessorUsageExpression

`ExpectedProcessorUsageExpression` defines the expected processor usage of a capability in terms of the number of processors needed for the calculation. For a single-threaded service, the `ExpectedProcessorUsageExpression` formula is constant `'1'`, which means that it uses a single processor. For a fully multi-threaded service, `ExpectedProcessorUsageExpression` is `'ProcessorCount'`, which means that it uses all available processors. For a service that is partly single-threaded, and partly multi-threaded, the `ExpectedProcessorUsage` formula is `'1 + f * (ProcessorCount – 1)'`, which means that during its runtime, the capability uses an average of one processor plus a fraction (`f`) of the remaining available processors. This fraction is the multi-threading factor and can be between 0 and 1.

The number of processors that the capability can use for that particular calculation session is specified through

`ISessionProperties::GetNumberOfProcessorsToUse()`. Calling `GetNumberOfProcessorsToUse()` is only allowed when `ExpectedProcessorUsageExpression` contains the `ProcessorCount` variable.

ExpectedMemoryUsageInMBExpression

`ExpectedMemoryUsageInMBExpression` defines the expected memory usage of a capability in terms of the predefined variables. These variables are capability specific. For example, the variables can be used to specify, for instance, the sizes of the fields, calculation grid and CT image set. Unlike `ExpectedProcessorUsageExpression`, `ExpectedMemoryUsageInMBExpression` must be accurate or at least it should not underestimate the memory usage. If underestimation occurs, the DCF may schedule a capability to run on a machine with insufficient physical memory. This may cause this and other simultaneously running capabilities to fail for lack of memory and/or cause the machine to start swapping working set data between main memory (RAM) and the external storage, which can have a considerable performance impact on the calculation.

ResourceUsageHelperVariableDefinitions

`ExpectedMemoryUsageInMBExpression` can be quite complex because of its required accuracy and dependence on a potentially large number of variables. To make complex expressions easier to write, the expression can be written in terms of helper variables that are defined using `ResourceUsageHelperVariableDefinitions`. `ResourceUsageHelperVariableDefinitions` contains a list of variable definitions separated with semi-colons. Each variable definition is of the form `'variable-name = expression'`, in which `variable-name` is the name of the variable and `expression` is an expression written in terms of predefined variables. Prefix the names of resource usage helper variables with `H_`.

HostRequirementsExpression

Defining `HostRequirementsExpression` is optional. Define it to specify which kind of computer (or computing environment) the capability needs. If defined, each DCF Agent advertises the capability only if the expression evaluates to true in the computer the DCF Agent resides in. `HostRequirementsExpression` cannot be used to restrict from which computers the algorithm is allowed to be started and used.

Syntax of the Expressions

The estimates are defined using arithmetic expressions, which can also contain predefined variables. For a list of the available variables, see Chapter 11 "Resource Usage Estimate Expression Variable Reference" on page 96. The syntax and operators of the expressions is the same as for arithmetic expressions in C++, except for the following differences:

- The logical AND operator is `and`
- The logical OR operator is `or`
- The logical NOT (`!`) operator is not supported
- Power operator is `^` (it has the highest operator precedence)
- Literal strings are placed in single quotes.

It is also possible to define if-then-else expressions by using the following syntax: `if` *LogicalExpr* `then` *Expr* `else` *Expr*.

Example Expressions

- A few `ExpectedProcessorUsageExpressions`:
  - `1`
  - `1 + 0.2 * (ProcessorCount – 1)`
  - `ProcessorCount`
- A few `ExpectedMemoryUsageInMBExpressions`:
  - `200`
  - `100+NumberOfPoints*NumberOfFields/1000`
- A few `ResourceUsageHelperVariableDefinitions`:
  - `H_ImageSize = NumberOfSlices * ImageXSizeInPixels * ImageYSizeInPixels`
  - `H_CalculationAreaSize = (CalculationAreaXSizeInMM * CalculationAreaYSizeInMM * CalculationAreaZSizeInMM / (10*CalculationGridSize)^3)`
- A few `HostRequirementsExpressions`:
  - `ProcessorCount >= 4`
  - `HostName == ’ws1-with-gpu’ or HostName == ’ws2-with-gpu’`

Error Handling

EAAPI has one single way to report errors to Eclipse: the `PutErrorFlag` method of the capability-specific requester. `PutErrorFlag` signals to Eclipse that an error has occurred. `PutErrorFlag` is not available in the `EAAPI::MessageHelper` class, because the recommended approach to error handling is to centralize catching and reporting the error exceptions to the session handling method of the capability (to the callback methods of EAAPI).

Note *Any error condition in your algorithm should end up with a call to the* `PutErrorFlag` *method of the capability-specific requester (with a descriptive error message), after which your capability should exit.*

The recommended approach to error handling when using EAAPI is outlined below:

- Embed the code of the `Do<action>` methods of your capability inside a try-catch statement, so that you catch `const std::exception& e`.
- Whenever you catch the exception mentioned above (or its derivative) in the catch-block, signal an error to Eclipse by calling `requester.PutErrorFlag(e.what())` and let the `Do<action>` method of the capability to return without further processing.
- Whenever you detect an error condition in your algorithm/code, raise an exception that derives from `std::exception` (such as `std::runtime_error`) with a descriptive error message (in the local language).

Requests are expected to either succeed or fail, and your capability should be implemented to handle both scenarios. In the case of request failure

(`EAAPI::RequestException`), you can issue `std::runtime_error` with an appropriate error message in the local language. If your session handling method contains the try-catch mechanism described above, the error is reported to Eclipse, after which your servant will exit.

The example below shows the error handling mechanism described above:

```
void MyPhotonPlanVolumeDoseCapability::DoHandlePhotonPlanVolumeDoseSession(const
EAAPI::ISessionProperties& sessionProperties,
EAAPI::IPhotonFieldVolumeDoseRequester& requester)
{
	try {
		MyLocalizationEngine::Initialize(sessionProperties.GetLanguageId());

		requester.PutProgress(0.1, TR("Input data..."));

		std::auto_ptr<DCRVAPlan> planAutoptr;
		try {
			planAutoptr = requester.GetPlan110();
		} catch (const EAAPI::RequestException&) {
			throw std::runtime_error(TR("Invalid input data. Plan is
missing."));
		}

		std::vector<EAAPI::FieldNumberAndWeight> fieldNumberAndWeightList;
		requester.GetFieldNumberAndWeightList(fieldNumberAndWeightList);

		std::auto_ptr<DCRVAPlan> densityAutoptr;
		try {
			densityAutoptr = requester.GetDensity();
		} catch (const EAAPI::RequestException&) {
			throw std::runtime_error(TR("Invalid input data. Density image
is missing."));
		}

		std::auto_ptr<DCRVAPlan> structureSetAutoptr;
		try {
			structureSetAutoptr = requester.GetStructureSet();
		} catch (const EAAPI::RequestException&) {
			throw std::runtime_error(TR("Invalid input data. Structureset
is missing."));
		}

		std::string calculationOptionsXmlString;
		try {
			calculationOptionsXmlString =
requester.GetCalculationOptionsXmlString();
		} catch (const EAAPI::RequestException&) {
			throw std::runtime_error(TR("Invalid input data. Calculation
options are missing."));
		}

		requester.PutCalculationStartNotification();

		std::auto_ptr<DCRVADose> planDoseAndIMRTActualBeamFluencesAutoptr =
MyPlanDoseCalculator::Calculate(planAutoptr.get(), fieldNumberAndWeightList,
densityAutoptr.get(), structureSetAutoptr.get(), calculationOptionsXmlString);

		requester.PutCalculationFinishedNotification();

requester.PutPhotonPlanVolumeDoseFinalOutput110(*planDoseAndIMRTActualBeamFluenc
esAutoptr);

		requester.PutProgress(1.0, TR("Calculation finished."));
	} catch (const std::exception& e) {
		requester.PutErrorFlag(e.what());
	}
}
```

In the example, notice the following:

- Request exceptions are converted into `std::runtime_error` exceptions with descriptive error messages in the end-user's local language (the localization engine in the example is initialized in the local language of the Eclipse user, after which the TR function would return a translated text for the given key).
- Any `std::exception` derivative (including the `std::runtime_error` exception) is caught by the last catch statement, and catch block signals the error to Eclipse by using the `requester.PutErrorFlag` method, after which the capability returns.

Note *Do not raise* `EAAPI::RequestException` *in your own code.*

Retrieving Input Data and Sending Output Data

For more information on retrieving input data and sending output data, see Chapter 8 "Capability Reference" on page 52 and "Request Reference" on page 78.

Resolving the Beam Data Directory Path to Use

During execution, the custom algorithm must resolve the treatment machine ID / energy mode pairs (electrons and photons) or the treatment machine ID / treatment technique pairs (protons) in the input data to the corresponding beam data directory using the interfaces provided by the EAAPI, and use the resulting beam data directory path for finding the beam data for the calculation.

Note *A custom algorithm may not use any other directories for finding the beam data. The algorithm may not modify or cache the beam data content in any way.*

During execution, the algorithm must resolve the beam data directory path to use using the interfaces provided by EAAPI, as follows:

- Retrieve the beam data root directory path using the `GetBeamDataRootDirectoryPath()` request of the capability-specific requester.
- Instantiate an object of type `EAAPI::IBeamDataAccessor` using `EAAPI::BeamDataUtils::CreateBeamDataAccessor(beamDataRootDirectoryPath)`, in which `beamDataRootDirectoryPath` is the beam data root directory path received from the capability-specific requester.
- Resolve the treatment machine ID and energy mode (or treatment technique) string of the corresponding field or fields from the input plan. Further resolve the treatment machine ID and energy mode (or treatment technique) pair to the specific beam data directory path using the `EAAPI::IBeamDataAccessor::GetSpecificBeamDataDirectoryPath(treatmentMachineId, energyModeOrTreatmentTechnique)` method.

Character strings in DICOM may be padded with a trailing space (or with a null character). The same padding applies to data objects in EAAPI (for instance, treatment machine ID or plan ID). You can use the

`EAAPI::Utils::RemoveTrailingPadding` method to remove the trailing spaces and null characters before attempting to resolve the IDs to the specific beam data directories. In addition, test your algorithm using, for example, a treatment machine with an odd-numbered ID and verify that the capability correctly resolves the beam data directory to be used.

Note *Beam data is not supported for `PhotonOptimalFluenceToLeafMotionsCapability`, and the capability must be designed so that it does not need or use beam data.*

Reporting the Beam Data Directory Path Used by an Algorithm

Each capability must report to Eclipse the beam data directory used. This is to ensure that the relevant beam data directory path gets logged to the plan-specific calculation notes in Eclipse, which makes it available and identifiable later on. The capability can report the beam data directory it is using by calling `EAAPI::IBeamDataAcccessor::PutBeamDataDirectoryInformation(informationMessage)`, in which `informationMessage` describes the beam data directory the capability is using, for instance:

```
Beam data directory path for treatment machine 'Varian_23EX' and energy mode
'6X': '\\dcfs\dcf$\client\BeamData\API_MyPhotonDoseAlgorithm\006'.
```

If a capability does not use beam data, it must be explicitly stated by calling `EAAPI::BeamDataUtils::StateThatCapabilityDoesNotUseBeamData()`.

If neither `EAAPI::IBeamDataAccessor::PutBeamDataDirectoryInformation` nor `EAAPI::BeamDataUtils::StateThatCapabilityDoesNotUseBeamData` is called by the capability, the capability is not able to send final output back to Eclipse.

If none of the capabilities in the servant use beam data, the servant must state that by calling `EAAPI::Servant::SetDoesNotRequireBeamData(true)` in the main function of the servant before calling the `EAAPI::Servant::Main` method. In addition, each capability must still explicitly state they do not use beam data.

For the header file, see `include/EAAPI/IBeamDataAccessor.h`. For usage examples, see the provided example algorithms.

Sending Progress, Information, Warning and Debug Messages

Note *The authors of custom algorithms are responsible for verifying the accuracy of all custom messages generated by the custom algorithms that are stored, displayed and printed in Eclipse.*

Your capability can send progress, information, warning and debug messages using the capability-specific requester's `PutProgress`, `PutInformation`, `PutWarning` and `PutDebug` methods. For convenience, the same functionality is also available through the `EAAPI::MessageHelper` class. It is semantically a singleton class, and its methods are "static", so its methods can be used without having to instantiate it first.

`EAAPI::MessageHelper` does not contain the `PutErrorFlag` method, whereas the capability-specific requesters do. This is to promote centralizing the main error handling to the `Do<action>` methods of each capability. For more information, see Section "Error Handling" on page 36.

Calling `PutProgress` notifies Eclipse of the current progress status of the servant. As a result, Eclipse updates the overall progress percentage and progress message of the currently running calculation tasks in the user interface. Since Eclipse may have divided the calculation into multiple sub-tasks, the value shown on the Eclipse progress bar can differ from the value sent from the individual servant.

Information, warning and error messages are collected and shown by Eclipse in the same categories in the Eclipse calculation dialog and saved to the calculation notes of the field.

Debug messages are special types of messages meant to aid in debugging or troubleshooting the algorithm. Debug messages are not stored to the calculation notes of the field, nor are they saved to the database.

Note *The* `PutDebug` *messages are available in Eclipse if the "Debug Output" option is set to 'on' in DCF Settings. If it is, the debug log is also available in the DCF log directory, usually* `C:\VMSOS\Log\Application\Vision\DCF`*.*

`PutInformation`, `PutWarning` *and* `PutError` *messages are automatically copied to the debug log by EAAPI. EAAPI also adds corresponding prefixes to the messages: "*`INFORMATION:`*", "*`WARNING:`*" and "*`error:`*". Moreover,* `PutDebug` *messages are prefixed with "*`DEBUG:`*" in the debug log of EAAPI.*

To detect whether the debug logging is currently enabled or disabled, the capability can use the `EAAPI::ISessionProperties::IsDebugLogOn` method. This is useful only in special cases, for instance, when the algorithm wants to avoid any overhead in the construction of the debug messages when the debug log is disabled.

Header Files

- `include/EAAPI/MessageHelper.h`
- `include/EAAPI/ISessionProperties.h`.

Sending Calculation Start and Calculation Finished Notifications

Your capability must notify Eclipse when it is starting calculation and when it has finished it. These notifications are automatically issued by EAAPI for `PhotonOptimizationCapability` and `ProtonOptimizationCapability`. For other capabilities, you must use the `PutCalculationStartNotification` and `PutCalculationFinishedNotification` methods.

Your algorithm must issue the `PutCalculationStartNotification` and `PutCalculationFinishedNotification` calls before it can send final output to Eclipse. Otherwise, EAAPI prevents sending the final output (by issuing an exception).

Localization

The `Do<action>` method of each capability (such as the `EAAPI::IPhotonFieldVolumeDoseCapability::DoHandlePhotonFieldVolumeDoseSession` method) has a `sessionProperties` parameter, which is of type `EAAPI::ISessionProperties`. Your capability can query the language of the current Eclipse user with method

`EAAPI::ISessionProperties::GetLanguageId`. The return value indicates the language in which your capability should operate. For instance, the progress, information, warning and error messages should be in that language. The messaging-related methods of EAAPI expect UTF-8 encoded strings (of type `std::string`). The character set for the C++ project has to be set to “Use Multi-Byte Character Set”.

If your capability does not support the corresponding language, it should exit with an error.

Note *The authors of custom algorithms are responsible for verifying the accuracy of all custom messages generated by the custom algorithms that are stored, displayed and printed in Eclipse.*

Header File

`include/EAAPI/ISessionProperties.h`

Adding Calculation Options to Your Algorithm

Calculation options enable you to let the user of your algorithm specify options for the algorithm. The user can specify the options in Eclipse, for each plan individually, before starting the calculation. Your algorithm can retrieve the options (specific for the current plan) by calling the `GetCalculationOptionsXmlString` method of the capability-specific requester. You determine the available calculation options based on what kind of options your algorithm needs or supports. Usually the calculation options modify the algorithm behavior affecting the dose results.

Calculation option definitions are servant-specific—each capability in the servant receives the same calculation options. Depending on the implementation of your algorithm, each capability can, however, use different parts of the calculation options.

Each algorithm must have calculation options, even if it does not use them. If your algorithm does not use calculation options, define empty but valid options schema XSD and default options XML files and place them into the DCF directory.

Calculation options are defined using XML. The calculation options schema is defined using an XSD file. The XSD file defines the available options and the valid value range of the options. The default values (which adhere to the schema) are defined using an XML file.

In the calculation options schema:

- The start tag of an element defines a node in the option tree (the node is visible to the user in the Eclipse calculation options dialog).
- Elements can be nested to achieve subnodes (tree-like options structure).
- The attributes of an element define the actual options that can be edited by the user, and the values of your algorithm can receive.
- The content of the elements is ignored.

For examples of the options schema XSD and default options XML, see the provided example algorithms.

The XSD and XML files must be placed to the `<DCF>\client\Options ([API])` directory. The XSD and XML filenames to be used are determined by the EAAPI based on the name and version of the algorithm. To see which filenames to use for your calculation options files, execute the servant executable from the command line with the `generate-configuration-info` argument (note that `VCServant.dll` or `VCServantd.dll` must be in the same directory with the servant executable) and inspect the generated `AlgorithmInfo.xml` (generated in the same directory as the servant executable).

The `EAAPI::Utils::ConvertCalculationOptionsXmlStringToMapOfStrings` method converts the calculation options XML string to a map of `<optionPath,optionValue>` pairs. It is recommendable to implement your algorithm to check that:

- It receives each option it knows about.
- There are no unrecognized options in the input.

Special Considerations

Stdin/stdout/stderr are Reserved Exclusively for EAAPI

Because EAAPI uses `stdin`, `stdout` and `stderr` streams for internal communication purposes, your servant (or the libraries it uses) may not directly write to those streams to avoid interfering the communication protocol.

Outputting and Displaying Dates and Times

If your algorithm or the related configuration programs output, display or request date or time information, create the output in an unambiguous format to avoid any possibilities to misinterpret the information. Output the date in the long format, with the month spelled out and the year with four digits.

Log Messages and Removal of Duplicate Message Content

When you send messages to Eclipse using `PutInformation` and `PutWarning`, Eclipse removes any duplicate lines from the calculation notes, retaining the first line in the log with the specific content.

About Caching of Data

Custom caching of data (for example, for the consequent runs of the algorithm) is not supported.

Eclipse Algorithm API and Thread Safety

EAAPI's methods of the capability-specific requesters (and the methods of `EAAPI::MessageHelper`) are thread-safe, meaning that EAAPI synchronizes requests from multiple threads. The synchronization is implemented with a Critical Section object to make other threads in the same process wait for their turn while request-related communication is in progress. Other EAAPI methods, or objects, are not synchronized, so you need to provide any synchronization if applicable for your algorithm.

About Assertions and Your Code

Consider asserting any assumptions you make in your code to get any discrepancies detected as soon as possible.

Chapter 4 Installing and Uninstalling a Custom Algorithm

Installing a Custom Algorithm

When you want to test your algorithm, install it in the DCF and Eclipse.

Before starting the installation:

- Check that you have a compatible Eclipse (including the DCF) properly installed and configured.
- Ensure that your algorithm compiles correctly.
- Ensure that the binaries of your algorithm servant (including `VCServant.dll` or `VCServantd.dll`) are collected under one single self-contained directory to make the contents of the directory ready to be copied as-is to the DCF directory. During development, this can usually be conveniently achieved using the post-build step in Visual Studio.
- Ensure that you have valid Options Schema XSD and Default Options XML files defined for your algorithm, and that their filenames match your algorithm version.

Note *The DCF distributor can schedule the algorithm to be run on any DCF-connected workstation. We recommend making the servant and its binaries as self-contained as possible.*

Installing custom algorithm takes place in the following steps:

- Copy the servant binaries to the DCF directory.
- Generate configuration info for the algorithm.
- Register the servant to the `VCConfiguration.xml`.
- Register the algorithm to the `InstalledAlgorithms.xml`.
- Copy the Calculation Options XSD and XML files to the DCF directory.
- Instruct the DCF Agents to reload the configuration.
- Import the physical material table, if applicable for your algorithm, to the ARIA database.
- Create and configure a calculation model for your algorithm.

Note *The configuration files of the DCF may not be edited manually. Exercise caution when installing algorithms and pay attention to any error messages during the installation.*

To Install a Custom Algorithm

1. To copy the servant binaries to the DCF Directory, create a new directory below the DCF **`server`**`\bin` directory for the servant binary files.

   Prefix the directory names with `[API]` *and include the algorithm name and version, for example:* `<DCFDirectory>\server\bin\[API] My Algorithm (0.0.1)`.

2. Copy the binaries of your algorithm (including `VCServant.dll` or `VCServantd.dll`) to the new DCF directory.

3. To generate configuration info for the algorithm, issue the following command from the Command Prompt:

```
C:\> "<DCFDirectory>\server\bin\[API] My Algorithm (0.0.1)\MyServant.exe"
generate-configuration-info
```

   As a result, the directory of the servant executable should now contain up-to-date `ServantInfo.xml` and `AlgorithmInfo.xml` files. If your algorithm has defined a physical material table (using `EAAPI::Servant::SetPhysicalMaterialTable`), the directory of the servant executable should now also contain the physical material table in XML format, in the file `PhysicalMaterialTableInfo-[API] <PhysicalMaterialTableId>.xml`.

4. To register the servant to the `VCConfiguration.xml`, issue the following command from the Command Prompt:

```
C:\> <DCFDirectory>\server\bin\VCConfig.exe register-servant
VCConfigurationPath=<DCFDirectory>\server\VCConfiguration.xml
ServantXmlPath="<DCFDirectory>\server\bin\[API] My Algorithm
(0.0.1)\ServantInfo.xml"
```

5. To register the algorithm to `InstalledAlgorithms.xml`, issue the following command from the Command Prompt:

```
C:\> <DCFDirectory>\server\bin\InstallAlgorithm.exe install
"<DCFDirectory>\server\bin\[API] My Algorithm (0.0.1)\AlgorithmInfo.xml"
<DCFDirectory>\client\InstalledAlgorithms.xml
```

6. To copy the calculation options XSD and XML files to the DCF directory:

   a. Create the following Options directory to the DCF directory: `<DCFDirectory>\`**`client`**`\Options ([API])`

   b. Copy the options Schema XSD and Default Options XML files of your algorithm to the directory.

7. If you are using other service connector than `LocalServiceConnector`, the running DCF Agents need to be instructed to reload the configuration as follows: Login to the DCF Distributor machine, navigate to `http://localhost:57580`, select **Agents** > **Service status overview** and click **Instruct agents to reload configuration**.

8. If your algorithm has defined a physical material table, copy the `PhysicalMaterialTableInfo-[API] <PhysicalMaterialTableId>.xml` file to `<DCFDirectory>\client\PhysicalMaterialTables` directory. Then in RT Administration, go to **Clinical Data > Physical Materials**, and click **Import from DCF** button to import the physical material table(s) in the

`<DCFDirectory>\client\PhysicalMaterialTables` directory to the ARIA database.

Note *Custom physical material table imported to the ARIA database can be deleted only if it is not used by any plan that has a calculated dose.*

9. Create and configure a calculation model as instructed in *Beam Configuration Reference Guide*.
10. Test calculation with your algorithm.

Note *When you are working in an isolated development environment, it is not necessary to perform a complete uninstall/install of the algorithm (and configuration of the calculation model) each time. The new executable can be copied over the old one if the configuration of the algorithm (version, algorithm properties, and so on) has not been changed.*

Uninstalling a Custom Algorithm

Uninstalling a custom algorithm takes place in a reverse installation order.

To completely delete (except for the beam data) the custom algorithm from Eclipse and the DCF, take the following steps:

- Remove the associated calculation models.
- Delete the Calculation Options XSD and XML files.
- Deregister the algorithm from `InstalledAlgorithms.xml`.
- Deregister the servant from `VCConfiguration.xml`.
- Delete the custom algorithm binaries from the DCF directory.

To Uninstall a Custom Algorithm

Note *The configuration files of the DCF may not be edited manually. Exercise caution when uninstalling algorithms, and pay attention to any error messages during the uninstallation.*

1. Go to Beam Configuration and delete all the calculation models that are associated to the custom algorithm version you want to delete. For instructions, refer to *Beam Configuration Reference Guide*.
2. Delete the Options Schema XSD and Default Options XML files in directory `<DCFDirectory>\`**`client`**`\Options ([API])` that relate to the custom algorithm version you want to delete.
3. To deregister your algorithm from the `InstalledAlgorithms.xml` file, issue the following command from the Command Prompt:

```
C:\> <DCFDirectory>\server\bin\InstallAlgorithm.exe uninstall
"<DCFDirectory>\server\bin\[API] My Algorithm (0.0.1)\AlgorithmInfo.xml"
<DCFDirectory>\client\InstalledAlgorithms.xml
```

4. To deregister your algorithm from the `VCConfiguration.xml` file, issue the following command from the Command Prompt:

```
C:\> <DCFDirectory>\server\bin\VCConfig.exe deregister-servant
VCConfigurationPath=<DCFDirectory>\server\VCConfiguration.xml
ServantXmlPath="<DCFDirectory>\server\bin\[API] My Algorithm
(0.0.1)\ServantInfo.xml"
```

5. Delete the subdirectory below the DCF directory that contains the servant binaries, for instance `<DCFDirectory>`**`\server`**`\bin\[API] My Algorithm (0.0.1)`.

   The custom algorithm is now uninstalled from Eclipse and the DCF.

Chapter 5 Configuring User Rights for Custom Algorithms

User right *Use Eclipse API Algorithms* determines which user groups can use custom algorithms. After the default installation of Eclipse and the Platform Server, only user group *Service* has that user right enabled. If a user who does not belong to a user group with the *Use Eclipse API Algorithms* user right enabled attempts to calculate with a custom algorithm, Eclipse prevents the calculation and shows an error message.

To Configure User Rights for Custom Algorithms

1. In a web browser, log in to your Platform Server.
2. Go to **Security > Rights**.
3. In the **Permission Rights** list, locate **Use Eclipse API Algorithms**.
4. Enable the user right for the user groups that should be able to use custom algorithms, and disable the user right for the user groups that should not be able to use custom algorithms.

Consider creating a dedicated user group, such as "Researchers" or "API", for users allowed to use custom algorithms, and enable the Use Eclipse API Algorithms user right only for that user group.

Note *Depending on how you employ custom algorithms, make sure to give your personnel training as to what the custom algorithms are, how they are identified, what your practices regarding them are, and what the safety precautions and considerations related to them are.*

Chapter 6 Starting and Debugging Custom Algorithms

Configuring Eclipse to Calculate Locally

If you are at early stages of algorithm development, it is recommendable to configure your workstation to calculate locally without distributing the calculation tasks to other workstations. For instructions on how to change your workstation to calculate locally, refer to *Beam Configuration Reference Guide*.

To Calculate with Your Algorithm

1. Start Eclipse and log in with your user credentials.
2. Create a test patient and plan of the type required by your algorithm.
3. Go to **External Beam Planning**.
4. In the Info Window, select the **Calculation Models** tab.
5. Select the Calculation Model you created or configured for your algorithm.
6. Start the calculation.

Eclipse should perform the calculation with your algorithm. If you are using `LocalServiceConnector` in your Local DCF Settings, the calculation is done locally on your workstation; otherwise it is automatically distributed to the available workstations.

For detailed instructions on selecting and calculating with calculation models, refer to *Treatment Planning for External Beam, Eclipse Reference Guide*.

To Inspect the Debug Output of the Algorithm

- To inspect the debug output of your algorithm, go to the Local DCF Settings and set the Debug output configuration option to `true`.

  This will transmit debug messages (resulting from `PutInformation`, `PutWarning`, `PutError`, `PutDebug`) from your algorithm to Eclipse. The debug output is visible through the calculation dialogs which contain the Debug tab. The debug output is also available in the log files in the DFC log directory, usually `C:\VMSOS\Log\Application\Vision\DCF`.

To Debug the Servant Executable with a Debugger

Sometimes during the development you might want to inspect the execution of your algorithm using a debugger. You can use the following technique to attach a debugger to your servant process on demand:

1. Add the following code snippet to your capability's entry point (the example is for `PhotonFieldVolumeDoseCapability`):

```
#include <Windows.h>
#include "EAAPI/Utils.h"


void
MyPhotonFieldVolumeDoseCapability::DoHandlePhotonFieldVolumeDoseSession(const
EAAPI::ISessionProperties& sessionProperties,
EAAPI::IPhotonFieldVolumeDoseRequester& requester)
{
    if (EAAPI::Utils::IsControlAltShiftDown())
    {
        ::DebugBreak();
    }
    …
}
```

2. Compile and install the algorithm normally.
3. Copy the Program Database (`.pdb`) to the same directory with your servant executable in the DCF directory.
4. Ensure that you are using `LocalServiceConnector` so that the servant is executed on your workstation.
5. Start your algorithm from Eclipse. When the servant is about to launch, press CONTROL + ALT + SHIFT.
   This forces a breakpoint exception in your servant process. As a result, Windows will ask whether you to want to attach a debugger to the servant process.

Chapter 7 Example Algorithms

Eclipse Algorithm API contains example algorithms for the supported capabilities. The source code for the example algorithms is located in the `EAAPI\examples\source` directory. The prebuilt algorithms are located in the `EAAPI\examples\prebuilt` directory. It is recommendable to study how the example algorithms interpret the input data and populate the output data. The example algorithms also contain special test modes (through calculation options) which allow you to get acquainted with the error handling aspects of the API.

Chapter 8 Capability Reference

This section describes the capabilities supported by Eclipse Algorithm API, outlines how each capability works and describes the related input and output data.

ElectronFieldVolumeDoseCapability

`ElectronFieldVolumeDoseCapability` is used for the calculation of the volumetric dose distribution for a single electron field. The calculation is based on the information obtained from the patient anatomy, field angle related to the patient, beam energy, size of the electron applicator, and the shape and material of any blocks present in the field, and any bolus attached to the field. This information is transferred from Eclipse to the custom dose calculation algorithm through the EAAPI. User must create custom beam data, which describes the physical properties of a specific beam (machine type, energy mode, applicator type).

Header Files

- `include/EAAPI/IElectronFieldVolumeDoseCapability.h`
- `include/EAAPI/IElectronFieldVolumeDoseRequester.h`

Input

- `GetBeamDataRootDirectoryPath`
- `GetCalculationOptionsXmlString`
- `GetCalculationVolume`
- `GetDensity`
- `GetFieldNumber`
- `GetMonteCarloSubfieldCount`
- `GetMonteCarloSubfieldNumber`
- `GetPlan110`
- `GetStructureSet`

NOTICE **Bolus included in the plan structure set should only be taken into account in the dose calculation of a field if the field is linked to the bolus.**

Output

- `PutElectronFieldVolumeDoseFinalOutput`

The algorithm may also send the following intermediate output: progress percentage, progress message, information messages and warning messages. Information and warning messages are stored in the calculation notes of the field.

Considerations

- Subfield number and subfield count are available as input only when the algorithm supports Monte Carlo parallelization, that is, when `EAAPI::Servant::SetCanDoMonteCarloParallelization(true)` has been called.
- When the Monte Carlo parallelization is in use, it is the responsibility of the servant to create a unique random number generator seed for each job. Also, the servant needs to reduce the number of particle histories to be simulated based on the Subfield count.

Communication with Eclipse

The capability communicates with Eclipse through the `IElectronFieldVolumeDoseRequester` interface. When the implementation of

```
virtual void
IElectronFieldVolumeDoseCapability::DoHandleElectronFieldVolumeDoseSession(
const EAAPI::ISessionProperties& sessionProperties,
EAAPI::IElectronFieldVolumeDoseRequester& requester ) = 0;
```

is called, the user is given a reference to a requester object. The requester object can be used to provide input data for the capability and output data, information, debug, error and status messages for Eclipse.

Applicable Algorithm Properties

- `SetName(std::string)`
- `SetVersion(int, int, int, int)`
- `SetDescription(std::string)`
- `SetCanDoMonteCarloParallelization(bool)`
- `SetCanHandlePatientSupportDevice(bool)`
- `SetDoNotBurnAssignedCTValues(bool)`
- `SetMaxCalculationGridSizeX(int)`
- `SetMaxCalculationGridSizeY(int)`
- `SetMaxCalculationGridSizeZ(int)`
- `SetMaxDensityImageSizeX(int)`
- `SetMaxDensityImageSizeY(int)`
- `SetMaxDensityImageSizeZ(int)`
- `SetNeedsAllStructures(bool)`
- `SetNeedsSegmentModel(bool)`
- `SetSupportedMachineModels(std::string)`

Related Capabilities

- None.

Related Example Algorithm

- `MyElectronDoseServant`

PhotonOptimizationCapability

The `PhotonOptimizationCapability` interface provides a method to implement a user-specified photon fluence optimization algorithm. The optimization takes input from the client application, calculates the fluences for the fields iteratively, and sends the optimized fluences to the client on each iteration.

Note *The capability can only send the* optimal *fluences to the Eclipse client. The optimal fluences must be run through an LMC (Leaf Motion Calculation) algorithm to get the leaf motions.*

Header Files

- `include/EAAPI/IPhotonOptimizationCapability.h`
- `include/EAAPI/IPhotonOptimizationRequester.h`

Input

- `GetBeamDataRootDirectoryPath`
- `GetCalculationOptionsXmlString`
- `GetCalculationVolume`
- `GetDensity`
- `GetFieldNumber`
- `GetInitialOptimalFieldFluences`
- `GetOptimizationInfo`
- `GetPlan110`
- `GetStructurePointModule`
- `GetStructureSet`

NOTICE **Bolus included in the plan structure set should only be taken into account in the dose calculation of a field if the field is linked to the bolus.**

- **Constraint module**: If the optimization objectives are changed in the Eclipse client between iterations, the module contains the new constraints from the client.
- **Parameter module**: If the fluence smoothing in X/Y or the dose minimization weights are changed in the Eclipse client between iterations, the module contains the new values from the client.

Output

- **Structure DVHs** for all structures.
- **Structure objective function values** for all structures for all iterations. The structure objective function values from each iteration are kept in the memory of the computer and sent to the Eclipse client in each iteration.
- **Total objective function values** for all iterations. The total objective function values from each iteration are kept in the memory of the computer and sent to the Eclipse client in each iteration.
- **Fluences**: Optimal fluences, or the opening ratio matrices, that is, fluences divided by the lateral beam intensity profile. The profile has to be taken into account if the beam intensity profile deviates from a flat profile, because optimal fluences are input for the LMC algorithm. The capability must populate the field fluences as follows:

  a. `OriginX` and `OriginY` of each field fluence must comply with DICOM Compensator Position, DICOM Tag (`300A, 00EA`), in which this coordinate is the center of the first pixel. The capability may only modify the following data related to the fluences: grid size, position, pixel data, `PixelSpacingX`, `PixelSpacingY`.

  b. `PixelSpacingX` and `PixelSpacingY` of each field fluence must be 2.5 millimeters.

  c. Pixel data of each field fluence must consist of `floats` (4 bytes/pixel).
- **Field weights**: Used only if the field weights are optimized instead of field fluences.

Communication with Eclipse

`EAAPI::IPhotonOptimizationCapability` has the following virtual abstract methods, which you need to implement in your capability:

- `DoGetInput`
- `DoProcessOneIteration`

When the capability executes, EAAPI first calls your implementation of the `DoGetInput` method. `DoGetInput` should retrieve input data applicable for your algorithm. After `DoGetInput` returns, EAAPI calls `DoProcessOneIteration` in a loop. In `DoProcessIteration` your algorithm should do the following:

- Check whether there are new optimization objectives and/or parameters from Eclipse (the user can modify the optimization objectives and parameters in Eclipse during the optimization) and take them into account in the corresponding iteration and onwards.
- Optimize a single iteration and return the results back to Eclipse

The optimization can stop at any time, which means that each `DoProcessOneIteration` *can be the last one and potentially the final output. If you want to stop and exit the optimization, set the* `stopOptimizationAndExitServant` *to true, in which case the output of that iteration is still considered for the results.*

The capability communicates with Eclipse through an `EAAPI::IPhotonOptimizationRequester` interface. Both `DoGetInput` and

`DoProcessOneIteration` methods are given a reference to a requester object. The requester object can be used to retrieve input data for the capability, and to send information, debug, error and status messages for Eclipse. The capability sends output to Eclipse by populating the `finalOutput` object of the `DoProcessOneIteration` method.

Applicable Algorithm Properties

- `SetName(std::string)`
- `SetVersion(int, int, int, int)`
- `SetDescription(std::string)`
- `SetCanHandlePatientSupportDevice(bool)`
- `SetDoNotBurnAssignedCTValues(bool)`
- `SetMaxDensityImageSizeX(int)`
- `SetMaxDensityImageSizeY(int)`
- `SetMaxDensityImageSizeZ(int)`
- `SetNeedsAllStructures(bool)`
- `SetNeedsSegmentModel(bool)`
- `SetSupportedMachineModels(std::string)`

Related Example Algorithm

- `MyPhotonOptimizationServant`

PhotonOptimalFluenceToLeafMotionsCapability

Between an IMRT optimization and the dose calculation, an optimal fluence generated by `PhotonOptimizationCapability` has to be transformed into leaf motions. The dose calculation requires the leaf motions and jaw positions as input data.

The `PhotonOptimalFluenceToLeafMotionsCapability` interface provides a method to implement a user-specified leaf motion calculation that can be understood as a continuation of the optimization. The specific constraints of any MLC type can be embedded into the module that manages this capability.

The leaf motion calculation should take into account the constraints and the dosimetric properties of the selected MLC and machine, and produce a leaf sequence that can reproduce the input optimal fluence with a reasonable accuracy. Other optimization criteria, such as the total treatment time or the number of control points, can also be set. The output of the optimizer should be a leaf sequence that fulfills the machine and MLC constraints (such as the maximum leaf exposure and X- and Y-limits), and have leaf positions and jaws defined for every control point. Meterset weights should also be calculated, and eventually the effect of the collimator back scatter has to be taken into account. In addition to the optimal fluence, an initial guess of MU is provided by Eclipse client as input, since the optimization might need to take into account the leaf and jaw speed constraints defined as meter-per-second.

Header Files

- `include/EAAPI/IPhotonOptimalFluenceToLeafMotionsCapability.h`
- `include/EAAPI/IPhotonOptimalFluenceToLeafMotionsRequester.h`

Input

- `GetCalculationOptionsXmlString`
- `GetLMCInputDataWithOptimalFluences`

Output

- `PutPhotonOptimalFluenceToLeafMotionsOutput`

The algorithm may also send the following intermediate output: progress percentage, progress message, information messages and warning messages. Information and warning messages are stored in the field calculation notes.

Considerations

- IMRT field normalization depends directly on the `lostMUfactor` and `fieldIntensity` reported by `PhotonOptimalFluenceToLeafMotionsCapability`. Mistakes there would lead to erroneous field normalization.

Communication with Eclipse

The capability communicates with Eclipse through an `IPhotonOptimalFluenceToLeafMotionsRequester` interface. When the implementation of

```
virtual void
IPhotonOptimalFluenceToLeafMotionsCapability::DoHandlePhotonOptimalFluenceToLeaf
MotionsSession( const EAAPI::ISessionProperties& sessionProperties,
EAAPI::IPhotonOptimalFluenceToLeafMotionsRequester& requester ) = 0;
```

is called, the user is given a reference to a requester object. The requester object can be used to provide input data for the capability, and output data, information, debug, error and status messages for Eclipse.

Applicable Algorithm Properties

- `SetName(std::string)`
- `SetVersion(int, int, int, int)`
- `SetDescription(std::string)`
- `SetDoesNotRequireBeamData(bool)`
- `SetSupportedMachineModels(std::string)`
- `SetSupportedMLCModels(std::string)`

Related Capabilities

- `PhotonLMCCBSFCapability`

Related Example Algorithm

- `MyPhotonLMCServant`

PhotonLMCCBSFCapability

If a custom photon dose calculation algorithm is intended to be used together with any Leaf Motion Calculator (LMC), excluding the Varian Leaf Motion Calculator (LMCV), `PhotonLMCCBSFCapability` (LMCCBSF, Leaf Motion Calculation Collimator Back Scatter Factor) needs to be used to generate leaf motions and actual fluence for IMRT fields. Eclipse uses this capability to get information about the radiation beam properties from the photon dose calculation algorithm, which is further transmitted to the LMC algorithm. This information is used by the LMC for the normalization of the actual fluence and for assuring that the leaf motions do not violate the machine limits.

Header Files

- `include/EAAPI/IPhotonLMCCBSFCapability.h`
- `include/EAAPI/IPhotonLMCCBSFRequester.h`

Input

- `GetBeamDataRootDirectoryPath`
- `GetCalculationOptionsXmlString`
- `GetPlanForPhotonLMCCBSFCapability`

Output

- `PutPhotonLMCCBSFFinalOutput`

The algorithm may also send the following intermediate output: progress percentage, progress message, information messages and warning messages. The information and warning messages are stored in the field calculation notes.

Constraints and Limitations

- Leaf Motion Calculators provided by Varian have limitations regarding the supported MLCs and treatment machines. To make the custom dose calculator to support IMRT fields with all MLCs available, it might be necessary to also provide `PhotonLMCMonitorUnitsCapability` required by LMCV (Varian Leaf Motion Calculator).

Considerations

- The `PhotonLMCCBSFCapability` needs to be defined for any calculation model with the `PhotonFieldVolumeDoseCapability` defined. Hence, these two capabilities need to be bundled together in the same servant, if the servant is to be used for dose calculation for IMRT fields.
- In dose calculation for IMRT fields, the MU are calculated by the Eclipse client instead of the calculation servant. To get correct MU, the calculated field dose distribution should be divided by a factor f_{norm} in the calculation servant. The normalization factor is defined in a following way:

  $f_{norm} = (MUGy_{ref} / MUGy(10,10) * D_{ref}$,

  where $MUGy_{ref}$ is the known ratio of MU to absorbed dose in Gy in a reference geometry, `MUGy(10,10)` is the known ratio of MU to absorbed dose in Gy in a 10 × 10 field, and D_{ref} is the calculated dose in the same reference geometry (in internal units of the algorithm). For IMRT calculation, the function `SetMUPerGray(double)` of class `VAOutputOptions` should be called with argument equal to zero, since Eclipse handles the MU calculation.

Communication with Eclipse

The capability communicates with Eclipse through the `IPhotonLMCCBSFRequester` interface. When the implementation of

```
virtual void IPhotonLMCCBSFCapability::DoHandlePhotonLMCCBSFSession(const
EAAPI::ISessionProperties& sessionProperties, EAAPI::IPhotonLMCCBSFRequester&
requester) = 0;
```

is called, the user is given a reference to a requester object. The requester object can be used to provide input data for the capability, and output data, information, debug, error and status messages for Eclipse.

Applicable Algorithm Properties

- `SetName(std::string)`
- `SetVersion(int, int, int, int)`
- `SetDescription(std::string)`
- `SetCanCalculateFFF (bool)`
- `SetCanCalculateSRS(bool)`
- `SetSupportedMachineModels(std::string)`

Related Capabilities

- `PhotonLMCMonitorUnitsCapability`
- `PhotonFieldVolumeDoseCapability`

Related Example Algorithm

- `MyPhotonDoseServant`

PhotonLMCMonitorUnitsCapability

If a custom photon dose calculation algorithm is to be used together with the Varian Leaf Motion Calculator (LMCV) to generate leaf motions and actual fluences for IMRT fields, a `PhotonLMCMonitorUnitsCapability` needs to be defined. Eclipse uses this capability to get information about the radiation beam properties from the photon dose calculation algorithm, which is further transmitted to the LMCV algorithm. This information is used by LMCV to get the correct scale for the actual fluence and to make sure that the leaf motions do not violate the machine limits.

Header Files

- `include/EAAPI/IPhotonLMCMonitorUnitsCapability.h`
- `include/EAAPI/IPhotonLMCMonitorUnitsRequester.h`

Input

- `GetBeamDataRootDirectoryPath`
- `GetCalculationOptionsXmlString`
- `GetPlanForPhotonLMCMonitorUnitsCapability`

Output

- `PutPhotonLMCMonitorUnitsFinalOutput`

The algorithm may also send the following intermediate output: progress percentage, progress message, information messages and warning messages. The information and warning messages are stored in the field calculation notes.

Constraints and limitations

- The Varian Leaf Motion Calculator (LMCV) can be used for the IMRT calculation only for Varian treatment units.

Considerations

- `PhotonLMCMonitorUnitsCapability` needs to be defined for any calculation model with the `PhotonFieldVolumeDoseCapability` defined. Hence, these two capabilities need to be bundled together in the same servant, if the servant is intended to be used for dose calculation for IMRT fields.
- In dose calculation for IMRT fields, the MU are calculated by the Eclipse client instead of the calculation servant. To get correct MU, the calculated field dose distribution should be divided by a factor f_{norm} in the calculation servant. The normalization factor is defined in a following way:

  $f_{norm} = (MUGy_{ref} / MUGy(10,10) * D_{ref}$,
- where $MUGy_{ref}$ is the known ratio of MU to absorbed dose in Gy in a reference geometry, `MUGy(10,10)` is the known ratio of MU to absorbed dose in Gy in a 10 × 10 field, and D_{ref} is the calculated dose in the same reference geometry (in internal units of the algorithm). For IMRT calculation, the function `SetMUPerGray(double)` of class `VAOutputOptions` should be called with argument equal to zero, since Eclipse handles the MU calculation.

Communication with Eclipse

The capability communicates with Eclipse through an IPhotonLMCMonitorUnitsRequester interface. When the implementation of

```
virtual void
IPhotonLMCMonitorUnitsCapability::DoHandlePhotonLMCMonitorUnitsSession( const
EAAPI::ISessionProperties& sessionProperties,
EAAPI::IPhotonLMCMonitorUnitsRequester& requester ) = 0;
```

is called, the user is given a reference to a requester object. The requester object can then be used to provide input data for the capability and output data, information, debug, error and status messages for Eclipse.

Applicable Algorithm Properties

- `SetName(std::string)`
- `SetVersion(int, int, int, int)`
- `SetDescription(std::string)`
- `SetCanCalculateFFF(bool)`
- `SetCanCalculateSRS(bool)`
- `SetSupportedMachineModels(std::string)`

Related Capabilities

- `PhotonLMCCBSFCapability`
- `PhotonFieldVolumeDoseCapability`

Related Example Algorithm

- `MyPhotonDoseServant`

PhotonFluenceCapability

- The implementation of `PhotonFluenceCapability` should be combined to same algorithm with `PhotonFieldVolumeDoseCapability` or `PhotonPlanVolumeDoseCapability` and it serves three functions:
    - Eclipse recognizes the dose calculation algorithm as compatible with EAAPI 11.0 based on the existence of `PhotonFluenceCapability`. If the capability is missing, Eclipse sends input data that is compatible with EAAPI 10.0. For that purpose, it suffices to implement the declaration of the capability.
    - Portal dose image prediction requires a fluence calculated for a part of an arc field or an IMRT field. The client uses `PhotonFluenceCapability` from the selected photon volume dose calculation model.
    - When large IMRT fields are split during plan approval, `PhotonFluenceCapability` is used to produce the actual fluences for split fields.

Header Files

- `include/EAAPI/IPhotonFluenceCapability.h`
- `include/EAAPI/IPhotonFluenceRequester.h`

Input

- `GetBeamDataRootDirectoryPath`
- `GetCalculationOptionsXmlString`
- `GetPlanForPhotonLMCCBSFCapability`

Output

- `PutPhotonFluenceFinalOutput`

The algorithm may also send the following intermediate output: progress percentage, progress message, information messages and warning messages. The information and warning messages are stored in the field calculation notes.

Considerations

- PhotonFluenceCapability should be bundled together with PhotonFieldVolumeDoseCapability or PhotonPlanVolumeDoseCapability as described above.
- PhotonFluenceCapability has to calculate the fluence for all fields in the input plan. The fluence itself should have opening-ratio-matrix normalization (that is, unity corresponds that the pixel is open during the entire radiation sequence). In addition an additional IMRT fluence normalization factor should be calculated from ImrtMUcoeff given by client. The actual fluence normalization is obtained by dividing ImrtMUcoeff by the MuGy(10,10) factor and multiplying it by the correction factor from Collimator back scatter factor.
- The fluence resolution should be the same as used by the dose calculation algorithm internally.
- The fluence should be calculated to a rectangular area covering at least the entire beam, and the center of the fluence should always be located at the central axis of the beam.

Communication with Eclipse

The capability communicates with Eclipse through an IPhotonFluenceRequester interface. When the implementation of

```
virtual void IPhotonFluenceCapability::DoHandlePhotonFluenceSession(const
EAAPI::ISessionProperties& sessionProperties, EAAPI::IPhotonFluenceRequester&
requester) = 0;
```

is called, the user is given a reference to a requester object. The requester object can then be used to provide input data for the capability and output data, information, debug, error and status messages for Eclipse.

Applicable Algorithm Properties

- SetName(std::string)
- SetVersion(int, int, int, int)
- SetDescription(std::string)
- SetSupportedMachineModels(std::string)
- SetSupportedMLCModels(std::string)

Related Capabilities

- PhotonFieldVolumeDoseCapability
- PhotonPlanVolumeDoseCapability

Related Example Algorithm

- MyPhotonDoseServant

PhotonFieldVolumeDoseCapability

`PhotonFieldVolumeDoseCapability` is used for the calculation of the volumetric dose distribution for a single photon field. The calculation is based on

- Information about the patient anatomy.
- Beam energy.
- Control point information containing field angles, collimator jaw positions, and leaf positions with corresponding meterset weights.
- Any compensator or block, with accompanying fluences. For compensators, the dose calculation may also be based on accompanying thickness information.
- Any wedges and bolus attached to the field.
- Machine and MLC IDs and the dosimetric properties of the MLC.
- For arc fields, the way the user has requested the approximation of the field.
- For IMRT fields, the IMRT normalization factor.
- Physical material table of the structure set (if supported by the algorithm).

Couch structures are included in the structure set and should be used for all fields in the plan. Bolus structures are also included in the structure sets. However, bolus may not be linked to all fields in the plan.

NOTICE **Bolus included in the plan structure set should only be taken into account in the dose calculation of a field if the field is linked to the bolus.**

This information is transferred from Eclipse to the custom dose calculation algorithm through the API. The custom beam data created for the algorithm should describe the physical properties of a specific beam (machine type, energy mode, applicator type).

Note *It is mandatory to implement* `PhotonFluenceCapability` *to the same calculation servant as* `PhotonFieldVolumeDoseCapability`*. Otherwise, the Eclipse client will erroneously consider the algorithm as a pre-EAAPI-11.0 version algorithm and attempt to provide wrong input data.*

Header Files

- `include/EAAPI/IPhotonFieldVolumeDoseCapability.h`
- `include/EAAPI/IPhotonFieldVolumeDoseRequester.h`

Input

- GetBeamDataRootDirectoryPath
- GetCalculationOptionsXmlString
- GetCalculationVolume
- GetDensity
- GetFieldNumberAndParams
- GetNextControlPointNumberToCalculate
- GetPhysicalMaterialTable
- GetPlan110
- GetStructureSet

Output

- PutPhotonFieldVolumeDoseFinalOutput110

In addition, the algorithm defines an MUperGray value, which is stored in the same dose object. Eclipse uses the MUperGray to convert the relative dose into absolute dose in units of Gy for a fixed number of monitor units. If the dose in Gy is fixed at a certain location in Eclipse, the required number of MU is computed by Eclipse from this information.

For IMRT fields, the `MUPerGray` value should not be defined in the algorithms. Instead, the dose should be normalized so that the relative dose matrix is normalized to `IMRTnormalization` (for more information, see `PhotonLMCCBSFCapability`) Since the MU for IMRT fields are precalculated to obtain the desired dose level, the dose has to be calculated corresponding to the precalculated MU. The client sends an additional `ImrtMU` coefficient for all IMRT fields that should be taken into account in the dose level.

An actual fluence should be sent to client for IMRT fields. For details, see `PhotonFluenceCapability`.

The algorithm may also send the following intermediate output: progress percentage, progress message, information messages and warning messages. Information and warning messages are stored in the field calculation notes.

Communication with Eclipse

The capability communicates with Eclipse through an `IPhotonFluenceRequester` interface. When the implementation of

```
virtual void
IPhotonFieldVolumeDoseCapability::DoHandlePhotonFieldVolumeDoseSession( const
EAAPI::ISessionProperties& sessionProperties,
EAAPI::IPhotonFieldVolumeDoseRequester& requester ) = 0;
```

is called, the user is given a reference to a requester object. The requester object can be used to provide input data for the capability and output data, information, debug, error and status messages for Eclipse.

Applicable Algorithm Properties

- `SetName(std::string)`
- `SetVersion(int, int, int, int)`
- `SetDescription(std::string)`
- `SetCanCalculateFFF(bool)`
- `SetCanCalculateMotorizedWedge(bool);`
- `SetCanCalculateOmniWedge(bool)`
- `SetCanCalculateSeparateControlPoints(bool);`
- `SetCanCalculateSRS(bool)`
- `SetCanCalculateVMAT(bool)`
- `SetCanHandleNonStandardFluenceResolution(bool)`
- `SetCanHandlePatientSupportDevice(bool)`
- `SetDoNotBurnAssignedCTValues(bool)`
- `SetMaxCalculationGridSizeX(bool)`
- `SetMaxCalculationGridSizeY(bool)`
- `SetMaxCalculationGridSizeZ(bool)`
- `SetMaxDensityImageSizeX(bool)`
- `SetMaxDensityImageSizeY(bool)`
- `SetMaxDensityImageSizeZ(bool)`
- `SetNeedsAllStructures(bool)`
- `SetNeedsSegmentModel(bool)`
- `SetSupportedMachineModels(std::string)`
- `SetCanCalculateFFF(std::string)`

Related Capabilities

- `PhotonPlanVolumeDoseCapability`
- `PhotonLMCCBSFCapability`
- `PhotonLMCMonitorUnitsCapability`
- `PhotonFluenceCapability`

Related Example Algorithm

- `MyPhotonDoseServant`

PhotonPlanVolumeDoseCapability

`PhotonPlanVolumeDoseCapability` is used for the calculation of the volumetric dose distribution for multiple photon fields. The calculation is based on:

- Information about the patient anatomy.
- Beam energy.
- Relative MU weights of the fields.
- Control point information containing field angles, collimator jaw positions, and leaf positions with corresponding meterset weights.
- Any compensator or block, with accompanying fluences.
- Any wedges and bolus attached to the field.
- Machine and MLC IDs and the dosimetric properties of the MLC.
- For arc fields, the way the user has requested the approximation of the field.
- For IMRT fields, the IMRT normalization factor.
- Physical material table of the structure set (if supported by the algorithm).

NOTICE **Bolus included in the plan structure set should only be taken into account in the dose calculation of a field if the field is linked to the bolus.**

This information is transferred from Eclipse to the custom dose calculation algorithm through the API. The custom beam data created for the algorithm should describe the physical properties of a specific beam (machine type, energy mode, applicator type).

Note *It is mandatory to implement* `PhotonFluenceCapability` *to the same calculation servant as* `PhotonPlanVolumeDoseCapability`*. Otherwise, the Eclipse client will erroneously consider the algorithm as a pre-EAAPI-11.0 version algorithm and attempt to provide wrong input data.*

Header Files

- `include/EAAPI/IPhotonPlanVolumeDoseCapability.h`
- `include/EAAPI/IPhotonPlanVolumeDoseRequester.h`

Input

This capability can use the following initial input for the dose calculation:

- `GetBeamDataRootDirectoryPath`
- `GetCalculationOptionsXmlString`
- `GetCalculationVolume`
- `GetDensity`
- `GetFieldNumberAndWeightList`
- `GetPhysicalMaterialTable`
- `GetPlan110`
- `GetStructureSet`

Output

The final output of the capability is the following:

- `PutPhotonFieldVolumeDoseFinalOutput110`

For IMRT fields, the `MUPerGray` value should not be defined in the algorithms. Instead, the dose should be normalized so that the relative dose matrix is normalized to `IMRTnormalization` (for more information, see `PhotonLMCCBSFCapability`) Since the MU for IMRT fields are precalculated to obtain the desired dose level, the dose has to be calculated corresponding to the precalculated MU. The client sends an additional `ImrtMU` coefficient for all IMRT fields that should be taken into account in the dose level.

An actual fluence should be sent to client for IMRT fields. See `PhotonFluenceCapability` for details.

The algorithm may also send the following intermediate output: progress percentage, progress message, information messages and warning messages. Information and warning messages are stored in the field calculation notes for all fields in the plan.

Considerations

- The calculation options for an algorithm that provides this capability must include the following option attribute: `PlanDoseCalculation ("ON"/"OFF")` (for definitions, refer to the options schema XSD and default options XML of the example algorithm). The algorithm must also provide a `PhotonFieldVolumeDoseCapability`.
- If the calculation option for the plan dose calculation is set `"ON"`, Eclipse uses this capability only if the following conditions apply; otherwise, `PhotonFieldVolumeDoseCapability` is used:
  - o The selected calculation algorithm for the volume dose offers `PhotonPlanVolumeDoseCapability`
  - o Plan has more than one field
  - o All fields in the plan use the same treatment unit, particle type, energy and primary fluence mode

- o None of the fields in the plan have Elekta Motorized Wedge or Elekta OmniWedge
- o Calculation has not been requested for a plan that contains reference points with fixed locations (Calculate Volume with Preset Values dialog box in the Eclipse client)
- o If there is a bolus, the same bolus is used in all fields. There is only the one bolus in the plan. No different bolus are used in the plan.
- o All fields in the plan are IMRT fields, or all fields in the plan are VMAT fields, or there are no IMRT fields and no VMAT fields in the plan.

Communication with Eclipse

The capability communicates with Eclipse through an `IPhotonPlanVolumeDoseRequester` interface. When the implementation of

```
virtual void
IPhotonPlanVolumeDoseCapability::DoHandlePhotonPlanVolumeDoseSession( const
EAAPI::ISessionProperties& sessionProperties,
EAAPI::IPhotonPlanVolumeDoseRequester& requester ) = 0;
```

is called, the user is given a reference to a requester object. The requester object can be used to provide input data for the capability and output data, information, debug, error and status messages for Eclipse.

Applicable Algorithm Properties

- `SetName(std::string)`
- `SetVersion(int, int, int, int)`
- `SetDescription(std::string)`
- `SetCanCalculateFFF(bool)`
- `SetCanCalculateSRS(bool)`
- `SetCanCalculateVMAT(bool)`
- `SetCanHandleNonStandardFluenceResolution(bool)`
- `SetCanHandlePatientSupportDevice(bool)`
- `SetDoNotBurnAssignedCTValues(bool)`
- `SetMaxCalculationGridSizeX(bool)`
- `SetMaxCalculationGridSizeY(bool)`
- `SetMaxCalculationGridSizeZ(bool)`
- `SetMaxDensityImageSizeX(bool)`
- `SetMaxDensityImageSizeY(bool)`
- `SetMaxDensityImageSizeZ(bool)`
- `SetNeedsAllStructures(bool)`
- `SetNeedsSegmentModel(bool)`
- `SetSupportedMachineModels(std::string)`
- `SetCanCalculateFFF(std::string)`

Related Capabilities

- `PhotonFieldVolumeDoseCapability`
- `PhotonLMCCBSFVolumeDoseCapability`
- `PhotonLMCMonitorUnitsCapability`
- `PhotonFluenceCapability`

Related Example Algorithm

- `MyPhotonDoseServant`

ProtonCompensatorCreationCapability

`ProtonCompensatorCreationCapability` is used for creating an ideal compensator that fulfils the plan parameters and constraints, such as the compensator resolution.

NOTICE **Eclipse does not support bolus structures for proton calculations. Ignore any bolus linked to a field in your algorithm. Furthermore, it is recommended to send a related calculation warning or to stop the algorithm with an error.**

Header Files

- `include/EAAPI/IProtonCompensatorCreationCapability.h`
- `include/EAAPI/IProtonCompensatorCreationRequester.h`

Input

- `GetBeamDataRootDirectoryPath`
- `GetCalculationOptionsXmlString`
- `GetCalculationVolume`
- `GetDensity`
- `GetFieldNumber`
- `GetIonPlan110`
- `GetStructureSet`

Output

- `PutProtonCompensatorCreationFinalOutput`

Communication with Eclipse

The capability communicates with Eclipse through an `IProtonCompensatorCreationRequester` interface. When the implementation of

```
virtual void
IProtonCompensatorCreationCapability::DoHandleProtonCompensatorCreationSession(
const EAAPI::ISessionProperties& sessionProperties,
EAAPI::IProtonCompensatorCreationRequester& requester) = 0;
```

is called, the user is given a reference to a requester object. The requester object can be used to provide input data for the capability and output data, information, debug, error and status messages for Eclipse.

Applicable Algorithm Properties

- `SetName(std::string)`
- `SetVersion(int, int, int, int)`
- `SetDescription(std::string)`
- `SetDoNotBurnAssignedCTValues(bool)`
- `SetNeedsAllStructures(bool)`
- `SetNeedsSegmentModel(bool)`
- `SetSupportedMachineModels(std::string)`

Related Capabilities

- `ProtonCompensatorConversionCapability`

Related Example Algorithm

- `MyProtonServant`

ProtonCompensatorConversionCapability

`ProtonCompensatorConversionCapability` is used for defining the closest milling pattern approximation of the ideal compensator calculated by `ProtonCompensatorCreationCapability`. The compensator conversion is not only intended to approximate the ideal compensator for the available milling hardware, but it should account for the milling machine parameters from the compensator calculation options. These options include, for instance, the drill bit size, drilling pattern, and compensator smoothing.

Header Files

- `include/EAAPI/IProtonCompensatorConversionCapability.h`
- `include/EAAPI/IProtonCompensatorConversionRequester.h`

Input

- `GetBeamDataRootDirectoryPath`
- `GetCalculationOptionsXmlString`
- `GetCalculationVolume`
- `GetFieldNumber`
- `GetIonPlan110`

Output

- `PutProtonCompensatorConversionFinalOutput`

Communication with Eclipse

The capability communicates with Eclipse through an `IProtonCompensatorConversionRequester` interface. When the implementation of

```
virtual void
IProtonCompensatorConversionCapability::DoHandleProtonCompensatorConversionSessi
on( const EAAPI::IsessionProperties& sessionProperties,
EAAPI::IProtonCompensatorConversionRequester& requester) = 0;
```

is called, the user is given a reference to a requester object. The requester object can be used to provide input data for the capability and output data, information, debug, error and status messages for Eclipse.

Applicable Algorithm Properties

- `SetName(std::string)`
- `SetVersion(int, int, int, int)`
- `SetDescription(std::string)`
- `SetSupportedMLCModels(std::string)`

Related Capabilities

- `ProtonCompensatorCreationCapability`.

Related Example Algorithm

- `MyProtonServant`

ProtonOptimizationCapability

Eclipse supports two kinds of treatment delivery modes for modulated scanning proton beams. These delivery modes are spot scanning and line scanning. EAAPI supports creating custom proton optimization algorithms for both of these delivery modes.

Proton Optimization can either be done as spot weight optimization or spot position and weight optimization. For spot weight optimization, the spot positions are fixed, and precalculated dose deposition coefficients (DDCs) can be used.

Both the spot weight optimization and spot position optimization types are supported for custom algorithms. Because the DDCs are calculated to fixed spot positions, algorithms using the DDCs can only perform the spot weight optimization. If the algorithm declares that it does not use DDCs, it is also permitted to modify the spot positioning. The DDCs are spot scanning specific, so line scanning optimization can only be done by algorithms that do not use the DDCs.

The energy layers are defined by the beamline calculation and modifications to the amount of energy layers is not supported.

NOTICE **Eclipse does not support bolus structures for proton calculations. Ignore any bolus linked to a field in your algorithm. Furthermore, it is recommended to send a related calculation warning or to stop the algorithm with an error.**

Header Files

- `include/EAAPI/IProtonOptimizationCapability.h`
- `include/EAAPI/IProtonOptimizationRequester.h`

Input

- `GetBeamDataRootDirectoryPath`
- `GetCalculationOptionsXmlString`
- `GetCalculationVolume`
- `GetDensity`
- `GetExpandedTargetSegment`
- `GetFieldNumber`
- `GetFieldNumberAndWeightList`
- `GetIonPlan110`
- `GetProtonDDCCachePoints`
- `GetSingleFieldOptimizationFlag`
- `GetStructurePointModuleProton`
- `GetStructureSet`
- `GetUniqueFractionation`

Intermediate Output

- Structure DVHs.
- Total objective function values for all iterations.

Output

- **Updated plan** with **control points** containing optimized spot weight values.

Capability Considerations

`ProtonOptimizationCapability` produces control points with optimized spot weights. Although you can also optimize for line scanning or continuous scanning, the results cannot be passed back to the Eclipse client and dose calculation. This means that the dose calculation algorithm must support the modulated spot scanning technique.

`ProtonOptimizationCapability` can use the Dose Deposition Coefficients (DDCs) as provided by the Proton Convolution Superposition (PCS) algorithm. DDCs are particular to Varian spot scanning optimization. They capture the non-weighted dose contribution from each scan spot to the structures. Structures are represented as point clouds in the optimization. As the majority of the DDCs are effectively zero, with the optimization, the problem can be formulated as a sparse linear system. Although the DDCs are made available to the capability, it does not have to use them. The example code demonstrates how the DDCs are retrieved from the cache files and how they are used to estimate the structure doses with the associated point clouds.

Communication with Eclipse

`EAAPI::IProtonOptimizationCapability` has the following methods which you need to implement in your capability:

- `DoGetInput`
- `DoProcessOneIteration`
- `DoReturnFinalOutput`

When the capability executes, EAAPI first calls your implementation of the `DoGetInput` method, which should retrieve input data applicable for your algorithm. After `DoGetInput` returns, EAAPI calls `DoProcessOneIteration` in a loop. In `DoProcessIteration` your algorithm should do the following:

- Check whether there are new optimization objectives from Eclipse (the user can modify the optimization objectives during the optimization) and take them into account in the corresponding iteration and onwards.
- Optimize a single iteration and return the intermediate results back to Eclipse.

When the final optimization results are needed, EAAPI calls `DoReturnFinalOutput`. EAAPI can also call `DoReturnFinalOutput` multiple times during the course of the optimization session (for instance, if the user resumes the optimization).

The optimization can stop at any time, which means that each `DoProcessOneIteration` and/or `DoReturnFinalOutput` can be the last one. Your algorithm can also momentarily interrupt the optimization by setting the value of `pauseOptimization` to `true` inside `DoProcessOneIteration`, in which case the output of that iteration is still considered for the results. `DoReturnFinalOutput` should be designed so that it always succeeds. If, however, a failure occurs, it must raise an exception.

Applicable Algorithm Properties

- SetName(std::string)
- SetVersion(int, int, int, int)
- SetDescription(std::string)
- SetCanHandlePatientSupportDevice(bool)
- SetDoesNotNeedPointCloudModel(bool)
- SetDoNotBurnAssignedCTValues(bool)
- SetMaxDensityImageSizeX(int)
- SetMaxDensityImageSizeY(int)
- SetMaxDensityImageSizeZ(int)
- SetNeedsAllStructures(bool)
- SetNeedsSegmentModel(bool)
- SetProtonOptimizationCapabilityUsesDDCCacheData(bool)
- SetProtonOptimizationSupportsNormalTissueConstraint(bool)
- SetSupportedMachineModels(std::string)

Related Capabilities

- `ProtonFieldVolumeDoseCapability`, which is automatically invoked once the optimization has been successfully completed and acknowledged by the user.

Related Example Algorithm

- `MyProtonServant`

ProtonFieldVolumeDoseCapability

`ProtonFieldVolumeDoseCapability` calculates an estimate for the dose distribution for a single field in a proton plan. The estimate is based on

- Initial beam characteristics at the nozzle entrance: nominal energy, beam shaping hardware in the nozzle and the final snout
- Treatment geometry with patient anatomy
- Treatment technique.

In a typical case, the capability is expected to calculate the dose for a completed plan, where the control point settings are no longer subject to change.

The beam shaping hardware depends on the treatment technique. It may consist of lateral spreaders, range shifters, range modulators, and deflection magnets, as well as information on the block and range compensator mounted on the final snout. Usually, the beamline modifier selection service of the PCS (Proton Convolution Superposition) algorithm can be used to pick the beamline hardware devices and their settings.

NOTICE **Custom algorithms must prevent dose calculation if the required input data (such as accessory, treatment machine, technique or energy) is unavailable or not supported.**

The block is defined in the Eclipse client. The block contour projected at the isocenter plane is passed in the beam container of the relevant field. The compensator can be calculated and converted with the PCS or, if the PCS does not model the beam line adequately, it can be calculated by `ProtonCompensatorCapability` and `ProtonCompensatorConversionCapability`. A static MLC can also be used to replace the block in Eclipse (Eclipse sends the MLC as DicomRTIONBeamLimitingDevice).

NOTICE **Eclipse does not support bolus structures for proton calculations. Ignore any bolus linked to a field in your algorithm. Furthermore, it is recommended to send a related calculation warning or to stop the algorithm with an error.**

Header Files

- `include/EAAPI/IProtonFieldVolumeDoseCapability.h`
- `include/EAAPI/IProtonFieldVolumeDoseRequester.h`

Input

- `GetBeamDataRootDirectoryPath`
- `GetCalculationOptionsXmlString`
- `GetCalculationVolume`
- `GetDensity`
- `GetFieldNumber`
- `GetIonPlan110`
- `GetStructureSet`

Output

- `PutProtonFieldVolumeDoseFinalOutput`

Considerations

The calculation options for an algorithm that provides `ProtonFieldVolumeDoseCapability` must include the following option attribute: `CreateVerificationPoints` (True/False) (for the definitions of the attribute, refer to the options schema XSD file and the default options XML file of the example algorithm). If the capability can output a verification point (that is, algorithm property `CanCalculateVerificationPointOnDoseCalculation` is `true`), and the user has set calculation option `CreateVerificationPoints` to `True`, Eclipse adds the verification point from `ProtonFieldVolumeDoseCapability` to the plan.

See also: `CanCalculateVerificationPointOnDoseCalculation` algorithm property, `PutProtonFieldVolumeDoseFinalOutput` request and *Treatment Planning for Proton Beam, Eclipse Reference Guide (Chapter 18, Section "About Verification Points"*).

Applicable Algorithm Properties

- `SetName(std::string)`
- `SetVersion(int, int, int, int)`
- `SetDescription(std::string)`
- `SetCanCalculateVerificationPointOnDoseCalculation(bool)`
- `SetCanHandlePatientSupportDevice (bool)`
- `SetDoNotBurnAssignedCTValues(bool)`
- `SetMaxDensityImageSizeX(int)`
- `SetMaxDensityImageSizeY(int)`
- `SetMaxDensityImageSizeZ(int)`
- `SetMaxCalculationGridSizeX(int)`
- `SetMaxCalculationGridSizeY(int)`
- `SetMaxCalculationGridSizeZ(int)`
- `SetNeedsAllStructures(bool)`
- `SetNeedsSegmentModel(bool)`
- `SetSupportedMachineModels (std::string)`

Related Capabilities

- `ProtonCompensatorCreationCapability`
- `ProtonCompensatorConversionCapability`

Related Example Algorithm

- `MyProtonServant`

Chapter 9 Request Reference

Table 4 Requests Supported by the Capabilities

Request	ElectronFieldVolumeDoseCapability	PhotonOptimizationCapability	PhotonLMCCBSFCapability	PhotonLMCMonitorUnitsCapability	PhotonOptimalFluenceToLeafMotionsCapability	PhotonFluenceCapability	PhotonFieldVolumeDoseCapability	PhotonPlanVolumeDoseCapability	ProtonCompensatorCreationCapability	ProtonCompensatorConversionCapability	ProtonOptimizationCapability	ProtonFieldVolumeDoseCapability
GetBeamDataRootDirectoryPath	■	■	■	■		■	■	■	■	■	■	■
GetCalculationOptionsXmlString	■	■	■	■	■	■	■	■	■	■	■	■
GetCalculationVolume	■	■					■	■	■	■	■	■
GetDensity	■	■					■	■	■		■	■
GetExpandedTargetSegment											■	
GetFieldNumber	■	■							■	■	■	■
GetFieldNumberAndParams							■					
GetFieldNumberAndWeightList								■			■	
GetInitialOptimalFieldFluences		■										
GetIonPlan110									■	■	■	■
GetLMCInputDataWithOptimalFluences					■							
GetMonteCarloSubfieldCount	■											
GetMonteCarloSubfieldNumber	■											
GetNextControlPointNumberToCalculate							■					
GetOptimizationInfo		■										
GetPhysicalMaterialTable							■	■				
GetPlan110	■	■					■	■				
GetPlanForPhotonFluenceCapability						■						
GetPlanForPhotonLMCCBSFCapability			■									
GetPlanForPhotonLMCMonitorUnitsCapability				■								
GetProtonDDCCachePoints											■	
GetSingleFieldOptimizationFlag											■	
GetStructurePointModule		■										
GetStructurePointModuleProton											■	
GetStructureSet	■	■					■	■	■		■	■
GetUniqueFractionation											■	
PutCalculationStartNotification	■		■	■	■	■	■	■	■	■		■
PutCalculationFinishedNotification	■		■	■	■	■	■	■	■	■		■
PutElectronFieldVolumeDoseFinalOutput	■											
PutPhotonFieldVolumeDoseFinalOutput110							■					
PutPhotonFluenceFinalOutput						■						
PutPhotonLMCCBSFFinalOutput			■									
PutPhotonLMCMonitorUnitsFinalOutput				■								
PutPhotonOptimalFluenceToLeafMotionsFinalOutput					■							
PutPhotonPlanVolumeDoseFinalOutput110								■				
PutProtonCompensatorConversionFinalOutput										■		
PutProtonCompensatorCreationFinalOutput									■			
PutProtonFieldVolumeDoseFinalOutput												■
PutErrorFlag	■	■	■	■	■	■	■	■	■	■	■	■
PutWarning	■	■	■	■	■	■	■	■	■	■	■	■
PutInformation	■	■	■	■	■	■	■	■	■	■	■	■
PutProgress	■	(■)	■	■	■	■	■	■	■	■	■	■
PutDebug	■	■	■	■	■	■	■	■	■	■	■	■

Table 5 Requests and Data Returned

Request	Data Returned
`GetBeamDataRootDirectoryPath`	**Directory root path** used to resolve the specific beam data directory to use. This is an input parameter to the `EAAPI::BeamDataUtils::CreateBeamDataAccessor` method. The algorithm should not modify the contents of the beam data directory in any way (it must consider the beam data directory as read-only).
`GetCalculationOptionsXmlString`	**Plan-specific calculation options** that the user can modify to control the dose calculation algorithm. `ElectronFieldVolumeDoseCapability`: For instance, the desired resolution of the 3D dose matrix, field normalization method, and use of heterogeneity correction. `PhotonFluenceCapability`: For instance, the desired fluence resolution. `PhotonOptimalFluenceToLeafMotionsCapability`: For instance, the desired number of control points, or the desired variation in sequencing technique. `PhotonOptimizationCapability`: For instance, whether the heterogeneity correction is used in the optimization, and default values for the fluence smoothing weights set in the optimization user interface. `PhotonFieldVolumeDoseCapability` and `PhotonPlanVolumeDoseCapability`: For instance, the desired resolution of the 3D dose matrix, field normalization method, and use of heterogeneity correction in the calculation. `ProtonCompensatorCreationCapability`: For instance, the desired resolution and compensator material properties, such as relative stopping power. `ProtonCompensatorConversionCapability`: For instance, the desired resolution. Ideal compensator(s) contained in the beam containers lack data on compensator material characteristics, such as the stopping power and milling machine parameters. This data can be passed in the calculation options or must be otherwise provided. *Care must be taken that compensator material stopping power, if used, is identical with the value used in compensator creation and subsequent dose calculation.* `ProtonOptimizationCapability`: For instance, the desired resolution and optimization method.
`GetCalculationVolume`	Rectangular parallelepiped which indicates the volume of interest in the image. The dose calculation algorithm usually computes the dose only to this volume to save computation time. However, Eclipse does not require that the output dose is only present in the calculation volume. If the resulting dose is smaller or larger than the calculation volume, the calculation volume is resized to the dimensions of the dose.

Request	Data Returned
`GetDensity`	The **size and resolution of the CT image**. The CT image can be transformed to the table-top coordinate system. Lookup curves are provided to interpret the CT image data as one of the following quantities: ▪ Electron density information ▪ Mass density information ▪ Proton stopping power ▪ HU values The CT image might have been acquired with a gantry tilt. Take this into account in your algorithm (see `DCRVADensity::IsTilted`). If your algorithm does not support images with a gantry tilt, it should exit with an error. *The input CT image is always in the treatment orientation selected for the corresponding plan. If your algorithm does not support the CT image orientation, it should exit with an error.*
`GetExpandedTargetSegment`	Returns a VASegmentVolume representation of the calculation target structure of a proton field. The target is expanded by the axial and lateral margins defined in the Eclipse client. The expansion is a similar operation to the Add Margin –tool in Contouring.
`GetFieldNumber`	**Field number**, which uniquely identifies the field for which the results are calculated, and refers to a field in the plan retrieved with the `GetPlan110` or `GetIonPlan110` requests. `ProtonOptimizationCapability`: only applicable if `GetSingleFieldOptimizationFlag` returns true.

Request	Data Returned
`GetFieldNumberAndParams`	**Field Number**, which uniquely identifies the field for which the results are calculated, and refers to a field in the plan retrieved with the `GetPlan110` request. **Dose matrix type** used only for when calculating the dose for the Elekta Motorized Wedge or the Elekta OmniWedge. In this case, two or three subdose matrices must be calculated: an open field, a field with the Motorized Wedge attached and a field with the Elekta Virtual Wedge. Each of these calculations requires a separate calculation with `PhotonFieldVolumeDoseCapability`. Eclipse uses this parameter to signal to the servant whether it should request a dose matrix of type `"OPEN"`, `"MWEDGE"`, `"VIRTUAL"` and `"FULL"`. The dose calculation should take into account the dose matrix type returned by the method as follows: ▪ `"OPEN"`–Ignore all wedge related modifiers in the plan object. Wedge modifiers should have weight 0.0. ▪ `"MWEDGE"`–Calculate the dose and take into account the wedge modifiers with weight 1.0 (the wedge covers the field completely). ▪ `"VIRTUAL"`–Calculate the dose with only the virtual modifiers. ▪ `"FULL"`–Calculate the dose taking all modifiers into account. **Use control point parallelization:** Used for conformal and dynamic arcs (but not for static arcs). If `true`, your algorithm must calculate and sum the dose for each control point queried from Eclipse. See also: `SetCanCalculateSeparateControlPoints` algorithm property.
`GetFieldNumberAndWeightList`	**Field numbers** and the **relative MU weights** of the fields for which the dose is calculated.
`GetInitialOptimalFieldFluences`	**Initial optimal field fluences**: The optimal fluences from the previous optimization; used for example after restarting the optimizer, and used as start fluences if available.

Request	Data Returned
`GetIonPlan110`	**Treatment plan** (`DCRVAPlan`) for the comprehensive treatment geometry (including treatment orientation) and the field-specific beam line configuration. For each planned field, the machine ID, the full geometry, and a set of control points (`DicomRTIONControlPoint`) are provided for detailed beam line configurations. ▪ `ProtonCompensatorConversionCapability`: The plan contains the ideal compensator (to be converted to milling pattern compensator by the algorithm) for the relevant field identified by `GetFieldNumber`. ▪ `ProtonFieldVolumeDoseCapability`, `ProtonOptimizationCapability`: For the modulated scanning technique with the PCS beam line selection, the set of control points specifies the scan spot coordinates and their weights. Setup fields are not passed as input to the algorithm. They are not included in the plan object. *Each field in the input plan has a unique field number. The field numbers are specific to the corresponding calculation session and can differ between calculation sessions.*

Request	Data Returned
`GetLMCInputDataWithOptimalFluences`	For each field to be calculated, the input is the following: **Optimal fluence matrix:** The requested fluence for radiation from the primary source. **Transient MU and dose rate:** Based on the current prescription and the MUGy10 factor, an estimate of the final MU is provided as input. The transient MU and dose rate are needed to verify that the jaw or leaf maximum speeds are not exceeded. **MLC and Machine model**: The same capability can be used for several different MLC/Machine combinations. The capability should recognize the supported MLCs and issue an error message for non-supported MLCs. To be able to handle certain MLCs, the servant should have a geometrical model of the MLC for the fluence calculation and the various constraints (such as interdigitation limitations or carriage motion restrictions). **Dosimetric data related to the MLC**: Leaf transmission factor and dosimetric leaf gap refers to the square leaf approximation but can be used also as dosimetric calibration parameters for more complicated leaf description. Leaf transmission factor and dosimetric leaf gap are set in Beam Configuration for each MLC material and energy mode. **Collimator back scatter table**: `PhotonLMCCBSFCapability` of the selected volume dose algorithm is called by the Eclipse client before leaf motion calculation to provide the collimator back scatter table. Collimator back scatter should be taken into account in the meterset weights of the output MLC sequence and the final MU. The meterset weights produced by the leaf motion calculation are used directly as the final meterset weights. No additional corrections are performed in the client. **MLC and relevant machine limits**: Various limits related to MLC or collimator, which can be set in Administration, such as constraints for the motion of the MLC leaves or jaws, or maximum number of control points. The servant should take these limitations into account when producing the MLC sequence. Once the MLC sequence is transferred back to the Eclipse client, it can be validated against the operating limits. **Field number:** Uniquely identifies the field to be calculated. Each field in the input plan has a unique field number.
`GetMonteCarloSubfieldCount`	**Monte Carlo subfield count:** The total number of subfields in the distributed calculation. A value greater than 1 indicates that Monte Carlo parallelization is in use.
`GetMonteCarloSubfieldNumber`	**Monte Carlo subfield number**: The index of a subfield in the range [1,..,subfield count]. Eclipse may divide the calculation of a single field into several jobs computed in parallel, typically at different workstations. Several jobs may also be simultaneously computed on a single workstation, if the workstation has sufficient computing power available. Each sub-job receives identical input with the exception of the subfield number, which is unique for each sub-job. The division into subfields is performed if the algorithm is capable of Monte Carlo parallelization and if the Monte Carlo parallelization factor is set to a value larger than 1 in the DCF settings.

Request	Data Returned
`GetNextControlPointNumberToCalculate`	**Next control point to calculate** if the control point parallelization is used for dynamic arcs, that is, `SetCanCalculateSeparateControlPoints(true)` has been called. If so, each servant is responsible for calculating the sum of doses from each control point.
`GetOptimizationInfo`	**Optimization info**: The optimization types (beamlet, that is, fluence optimization, field weight, or none) and the fixed jaw flags for the fields.
`GetPhysicalMaterialTable`	**Physical material table of the structure set**. Physical material table contains the following information for each physical material of the table: material ID, a minimum mass density, maximum mass density, and nominal (default) mass density. It is also indicated for each material whether the material is to be used in automatic conversion from mass density to material (it is the responsibility of the algorithm to perform the conversion). Other materials can only be assigned as structure materials, and their densities must be assigned within the allowed range. The physical material table received from the Eclipse client may differ from the physical material table defined by the algorithm. Eclipse may start the calculation if it considers the physical material table of the structure compatible with the physical material table defined by the algorithm. Two physical material tables are considered compatible by Eclipse in the context of a structure set with possible manual material assignments to structures, if the following conditions are met: ■ Both material tables have the same set of material IDs that can be automatically assigned ■ The minimum and maximum mass densities of all materials that can be automatically assigned are equal within the floating point accuracy and ■ Both tables agree that the manual mass density assignments to structures with manual material assignments are within the mass density limits of the corresponding materials. The algorithm is responsible for any additional compatibility checks (such as checking the exact equality). The algorithm can make this request only if it has a physical material table defined. See also SetPhysicalMaterialTable in “Table 7 Algorithm Properties” on page 92.

Request	Data Returned
`GetPlan110`	All the fields in a treatment plan for photons or electrons. Each field has information about the machine ID, energy mode, gantry angle, collimator rotation angle, couch rotation angle, any blocks or bolus attached to the field, MLC dosimetric parameters, and radiation type (photon or electron). Each field in the input plan has a unique field number amongst the fields in the plan. The field numbering is specific to the corresponding calculation session and can differ between calculation sessions. The plans also contain the following data, applicable only to certain capabilities: ▪ `ElectronFieldVolumeDoseCapability`: Applicator ID, which must be configured to uniquely define the applicator properties, such as the field size. ▪ `PhotonFieldVolumeDoseCapability`, `PhotonPlanVolumeDoseCapability`: wedges, compensators attached to the field. ▪ `PhotonFieldVolumeDoseCapability`, `PhotonPlanVolumeDoseCapability`, `PhotonFluenceCapability`: A field may be either static or arc. It may contain two control points (open field, static MLC field, standard arc) or multiple control points (IMRT field, conformal or VMAT arc). Any non-arc field with an IMRT normalization coefficient is considered an IMRT field. An IMRT field accompanies the `ImrtMUCoeff` factor that is required to obtain a correct IMRT normalization. ▪ `PhotonFieldVolumeDoseCapability`, `PhotonPlanVolumeDoseCapability`, `PhotonFluenceCapability`: Information required for fluence calculation: Machine and MLC type, dosimetric parameters of the MLC. Setup fields are not passed as input to the algorithm. They are not included in the plan object. The plan object contains the treatment orientation.
`GetPlanForPhotonFluenceCapability`	Photon fields requiring fluence calculation. Each field has information about the machine ID and energy mode (which can be used to determine the beam data directory specific for the energy mode) and MLC dosimetric parameters.
`GetPlanForPhotonLMCCBSFCapability`	Fields requiring LMC CBSF calculation. For each field, the applicable input is machine ID and energy mode, which can be used to determine the beam data directory specific for the energy mode.
`GetPlanForPhotonLMCMonitorUnitsCapability`	**Plan**: Contains only a single field in the plan, regardless of the number of fields in the plan. The capability should calculate the output data only for that field. The field has information about the machine ID and energy mode, which can be used to determine the beam data directory specific for the energy mode. The field also contains the jaw positions, based on which the collimator back scatter factor should be defined.

Request	Data Returned
`GetProtonDDCCachePoints`	Retrieves the cached DDC items for the specified field/structure combination. The DDC cache contains the pre-calculated point dose contribution coefficients calculated by the PCS algorithm. Each DDC item consists of the following: **DDC:** Dose Deposition Coefficient. Value of DDC is a fraction of the dose distributed with this energy to this point in the point cloud. The value is relative, that is, the sum of all the DDCs over all the energies is normalized to the prescribed total dose. Example. `DDC_i*TotalDose/SUM_i(DDC_i) = Dose_i`, where `DDC_i is DDC` in point `i`, `SUM_i` is the sum over all the (point cloud) points and energies, and `Dose_i` is the total dose. **Layer nominal energy:** Nominal energy of a layer. **Spot X:** X-coordinate of a spot in the coordinate system of the beam limiting device. **Spot Y**: Y-coordinate of a spot in the coordinate system of the beam limiting device. *Pre-calculated DDC cache items are only available if the algorithm has set the algorithm property* `ProtonOptimizationCapabilityUsesDDCCacheData` *to* `true`*. In this case, the DDC cache items are available only for those field/structure combinations in which the structure has objectives defined.*
`GetSingleFieldOptimizationFlag`	If `true`, the field number to be optimized can be retrieved with `GetFieldNumber`, otherwise all fields are optimized simultaneously.
`GetStructurePointModule`	**Structure point module**: Each structure is modeled as a set of points which are located randomly inside the structure; the data contain the coordinates of the points in the point cloud (x, y, z) in serial order. **Base dose:** If the base dose option has been selected by the user in the optimization dialog in the Eclipse client, the available dose of the corresponding plan to be taken into account by the algorithm.
`GetStructurePointModuleProton`	Structure point module containing point clouds generated by Eclipse. If cached DDCs are used, the optimization must operate with the contained point sets representing the structures. *The cached DDCs are coupled to point cloud representations of the structures.*

Request	Data Returned
`GetStructureSet`	**Structure set**: Includes the contour of the Body structure, and any bolus and couch structures in the plan using a segment volume representation. `PhotonFieldVolumeDoseCapability`, `PhotonPlanVolumeDoseCapability`, `ElectronFieldVolumeDoseCapability`: If the algorithm property `NeedsAllStructures` is `true`, all structures are included in the structure set. If the algorithm property `NeedsAllStructures` is `false` (the default), the structure set contains the structures as follows: ▪ `PhotonFieldVolumeDoseCapability`, `PhotonPlanVolumeDoseCapability`, `ElectronFieldVolumeDoseCapability`: Structures for bolus, Body and PTV (the target volume of the corresponding plan) are always included. ▪ `PhotonFieldVolumeDoseCapability`, `PhotonPlanVolumeDoseCapability`, `ElectronFieldVolumeDoseCapability`: If the algorithm property `CanHandlePatientSupportDevice` is `true`, the structures for the couch and other patient support devices are included, otherwise they are not included. ▪ `PhotonFieldVolumeDoseCapability`, `PhotonPlanVolumeDoseCapability`: If the algorithm property `DoNotBurnAssignedCTValues` is `true`, the structures with assigned HU values are included. If the algorithm property `DoNotBurnAssignedCTValues` is `false` (the default), the structures with assigned HU values are not included (they are burned to the density image). **Protons**: Structure set (`DCRVAStructureSet`) for segment volume data. This input includes the contour of the body, target and other segmented structures/organs as volume representations. For compensator calculation this information is largely redundant with distance matrix contained in field specific proton data structure (`DCRVAProtonData`). While provided structure set may be redundant if point clouds are used.
`GetUniqueFractionation`	Returns a pointer to an EAAPI::Fractionation –struct. The struct represents the fractionation information in Eclipse client. The struct contains the following attributes: ▪ `prescribedDosePerFraction` ▪ `prescribedPercentage` ▪ `planNormalizationFactor` Note: The pointer is valid only if a unique fractionation exists in Eclipse client.
`PutCalculationStartNotification`	Notifies Eclipse about the start of the actual calculation. Your capability (except `PhotonOptimizationCapability` and `ProtonOptimizationCapability`) must issue `PutCalculationStartNotification` and `PutCalculationFinishedNotification` requests (in that order) before it can send final output to Eclipse.

Request	Data Returned
`PutCalculationFinishedNotification`	Notifies Eclipse about the end of the calculation. Your capability (except `PhotonOptimizationCapability` and `ProtonOptimizationCapability`) must issue `PutCalculationStartNotification` and `PutCalculationFinishedNotification` requests (in that order) before it can send final output to Eclipse.
`PutElectronFieldVolumeDoseFinalOutput`	**Relative beam dose:** 3D matrix containing the calculated absorbed dose to the patient in relative units. In addition, the algorithm defines an `MUperGray` value, which is stored the same dose object. Eclipse uses the `MUperGray` to convert the relative dose into absolute dose in units of Gy for a fixed number of MU. If the dose in Gy is fixed at a certain location in Eclipse, the required number of MU is computed by Eclipse based on this information.
`PutPhotonFieldVolumeDoseFinalOutput110`	**Relative beam dose**: 3D matrix containing the calculated absorbed dose to the patient in relative units, possibly normalized based on the calculation options to the algorithm. In addition, the algorithm defines an `MUperGray` value (except for IMRT fields), which is stored in the same dose object. Eclipse uses the `MUperGray` to convert the relative dose into absolute dose in units of Gy for a fixed number of MU. If the dose in Gy is fixed at a certain location in Eclipse, the required number of MU is computed by Eclipse based on this information. For IMRT fields, the `MUperGray` value should not be calculated by the algorithm. Instead, the actual fluence containing the IMRT normalization should be included in the dose object. The actual fluence is needed for visualization and for the Varian PDC algorithm.
`PutPhotonFluenceFinalOutput`	**Actual beam fluencies** for each field received using `GetPlanForPhotonFluenceCapability` request. IMRT actual beam fluences should follow IMRT normalization.
`PutPhotonLMCCBSFFinalOutput`	The output must contain a field CBSF module for each field in the plan (for the plan returned by `GetPlanForPhotonLMCCBSFCapability`). Output for each field: **MUGy10x10:** The number of MU required to get the calculated dose of 1 Gy for a 10 cm × 10 cm open field at 10 cm depth at a source-to-phantom distance (SPD) of 100 cm. LMC uses this number only to check that the velocity constraints for dynamic collimators are not violated. **Collimator back scatter factor table:** A table of 41 double values containing the collimator back scatter factors (CBSFs) for symmetric, square field sizes in the range 0 ,.., 40 cm with the resolution of 1 cm. A CBSF factor defines the effect of back scatter from the collimators (jaws and MLC) to the total radiation output of the linac. The factor should be smaller than 1.0 for field sizes smaller than 10 cm × 10 cm and larger than 1.0 for field sizes larger than 10 cm × 10 cm. For the 10 cm × 10 cm field, the factor should be equal to 1.0.

Request	Data Returned
`PutPhotonLMCMonitorUnitsFinalOutput`	The output must contain a single `FieldMU` item for the field specified by `GetPlanForPhotonLMCMonitorUnitsCapability`. Output for the field: **Referenced field number:** Field number of the field for which the output is produced. The field number must be the same as the field number of the field in the plan retrieved with `GetPlanForPhotonLMCMonitorUnitsCapability`. **MUGy10x10:** The number of MU required to get the calculated dose of 1 Gy for a 10 cm × 10 cm open field at the depth of 10 cm at a source-to-phantom distance (SPD) of 100 cm. Varian Leaf Motion Calculator (LMCV) uses this number only to check that the velocity constraints for dynamic collimators are not violated. **Collimator back scatter factor:** Factor defining the effect of the back scatter from the collimators (jaws and MLC) to the total radiation output of the linac[1,2]. Needs to be defined for each field in the plan. Should be smaller than 1.0 for field sizes smaller than 10 cm × 10 cm, and larger than 1.0 for field sizes larger than 10 cm × 10 cm. For the 10 cm × 10 cm field, the factor should be equal to 1.0. *Calib3 and Calib4 are currently not used.*
`PutPhotonOptimalFluenceToLeafMotionsOutput`	**Dynamic MLC Sequence**: Optimized MLC sequence that reproduces the given optimal fluence with an acceptable accuracy and fulfills the operation limits and other constraints of the selected MLC and machine. Each control point should have the correct number of leaf coordinates and jaw positions. The collimator rotation or gantry rotation is not allowed by the Eclipse client. The meterset weights of the control points should be cumulative. If two consecutive control points have same meterset weights, a beam hold is assumed. Static segments are produced by having two consecutive control points that differ only by the meterset weights. **Actual fluence matrix**: Two-dimensional fluence matrix containing the contribution from both jaws and MLC leaves. The output actual fluence should follow the requested size and resolution. **MU factor**: Number describing the efficiency of the DMLC sequence. It is the ratio between the brightest fluence pixel in the optimal fluence, and the value of a pixel that would not be shielded by the MLC or the jaws during the whole sequence.

Request	Data Returned
`PutPhotonPlanVolumeDoseFinalOutput110`	**Relative plan dose**: 3D matrix containing the calculated absorbed dose to the patient from all fields in the plan in relative units, possibly normalized based on the calculation options set for the algorithm. In addition, the algorithm defines an MUperGray value (except for IMRT plan), which is stored in the same dose object. *The MUperGray value must be defined for each field in the plan.* Eclipse uses the MUperGray to convert the relative dose into absolute dose in units of Gy for a fixed number of MU. If the dose in Gy is fixed at a certain location in Eclipse, the required number of MU is computed by Eclipse based on this information. For IMRT fields, the `MUperGray` value should not be calculated by the algorithm. Instead, the actual fluence containing the IMRT normalization should be included in the dose object. The actual fluence is needed for visualization and for the Varian PDC algorithm.
`PutProtonCompensatorConversionFinalOutput`	**Updated plan** with the converted compensator. The beam container for the relevant field in the plan updated with the converted (milling pattern approximation) of the ideal compensator. The plan shall contain the converted compensator.
`PutProtonCompensatorCreationFinalOutput`	**Updated plan** with ideal compensator calculated based on the plan parameters, as constrained by the resolution and the available input data.
`PutProtonFieldVolumeDoseFinalOutput`	**Calculated field dose**: A 3D matrix containing the calculated absorbed dose to the patient in relative units, possibly normalized based on the calculation options to the algorithm. **Verification point:** If the algorithm property `CanCalculateVerificationPointOnDoseCalculation` is set to `true`, the output must also contain a verification point for the field.
`PutErrorFlag`	Sends an error message to Eclipse, signaling that an error has occurred during the execution of the algorithm/capability. After the servant has signaled an error using `PutErrorFlag`, it should exit without further processing.
`PutWarning`	Sends a warning message to Eclipse. EAAPI also copies the warning message to the debug log.
`PutInformation`	Sends an information message to Eclipse. EAAPI also copies the information message to the debug log.
`PutProgress`	Sends a progress message to Eclipse.
`PutDebug`	Sends a debug message to Eclipse (if debug output is enabled from Eclipse).

Chapter 10 Algorithm Property Reference

Table 6 Applicability of Algorithm Properties to Added Capabilities

Algorithm property	Default value	ElectronFieldVolumeDoseCapability	PhotonOptimizationCapability	PhotonLMCCBSFCapability	PhotonLMCMonitorUnitsCapability	PhotonOptimalFluenceToLeafMotionsCapability	PhotonFluenceCapability	PhotonFieldVolumeDoseCapability	PhotonPlanVolumeDoseCapability	ProtonCompensatorCreationCapability	ProtonCompensatorConversionCapability	ProtonOptimizationCapability	ProtonFieldVolumeDoseCapability
SetName	(empty)	■	■	■	■	■	■	■	■	■	■	■	■
SetVersion	(empty)	■	■	■	■	■	■	■	■	■	■	■	■
SetDescription	(empty)	■	■	■	■	■	■	■	■	■	■	■	■
SetCanCalculateCompensatorMUs	false							■	■				
SetCanCalculateFFF	false			■	■			■	■				
SetCanCalculateMotorizedWedge	false							■					
SetCanCalculateOmniWedge	false							■					
SetCanCalculateSeparateControlPoints	false							■					
SetCanCalculateSRS	false			■	■			■	■				
SetCanCalculateVerificationPointOnDoseCalculation	false												■
SetCanCalculateVMAT	false							■	■				
SetCanDoMonteCarloParallelization	false	■											
SetCanHandleNonStandardFluenceResolution	false							■	■				
SetCanHandlePatientSupportDevice	false	■	■					■	■			■	■
SetCanProtonOptimizeWithoutLowerConstraints	true											■	
SetDoesNotNeedPointCloudModel	false											■	
SetDoesNotRequireBeamData	false					■							
SetDoNotBurnAssignedCTValues	false	■	■					■	■	■		■	■
SetMaxCalculationGridSizeX	256	■						■	■				■
SetMaxCalculationGridSizeY	256	■						■	■				■
SetMaxCalculationGridSizeZ	1000	■						■	■				■
SetMaxDensityImageSizeX	256	■	■					■	■			■	■
SetMaxDensityImageSizeY	256	■	■					■	■			■	■
SetMaxDensityImageSizeZ	1000	■	■					■	■			■	■
SetNeedsAllStructures	false	■	■					■	■	■		■	■
SetNeedsSegmentModel	true	■	■					■	■	■		■	■
SetPhysicalMaterialTable	(none)							■	■				
SetProtonOptimizationCapabilityUsesDDCCacheData	false											■	
SetProtonOptimizationSupportsNormalTissueConstraint	false											■	
SetSupportedMachineModels	(empty)	■	■	■	■	■	■	■	■	■	■	■	■
SetSupportedMLCModels	(empty)					■	■	■	■				

Table 7 Algorithm Properties

Algorithm Property	**Default value**	**Description**
`SetName`	(empty)	Name of the algorithm.
`SetVersion`	(empty)	Version of the algorithm. *Only the first three elements (*`majorVersion`*,* `minorVersion` *and* `microVersion`*) are visible to the user of the algorithm. The last element (*`serviceRevision`*) affects the generated DCF service names and is not visible to the user in the Eclipse user interface.*
`SetDescription`	(empty)	Short description of the algorithm.
`SetCanCalculateCompensatorMUs`	false	Indicates that the algorithm can calculate MU for a field that contains a compensator.
`SetCanCalculateFFF`	false	Indicates that the algorithm can calculate dose for a field whose primary fluence mode is FFF (Flattening Filter Free).
`SetCanCalculateMotorizedWedge`	false	If set to true, defines that the algorithm can calculate the dose to fields that contain a motorized wedge accessory. Calculation of the MW requires that the algorithm can separately calculate the open and wedge parts of the field. Only meaningful for photon dose calculation algorithms.
`SetCanCalculateOmniWedge`	false	If set to true, defines that the algorithm can calculate dose to fields that contain an Elekta OmniWedge™ accessory. Calculation of the OmniWedge requires that the algorithm can separately calculate the open, wedge and motorized parts of the field. Only meaningful for photon dose calculation algorithms with virtual wedge.
`SetCanCalculateSeparateControlPoints`	false	Indicates whether the `PhotonFieldVolumeDoseCapability` can calculate control points as described in **Use control point parallelization** in `GetFieldNumberAndParams` request on page 81.
`SetCanCalculateSRS`	false	Indicates that the algorithm can calculate the dose for a field whose primary fluence mode is SRS (Stereotactic Radio Surgery).
`SetCanCalculateVerificationPointOnDoseCalculation`	false	Indicates whether the algorithm can calculate verification point during the dose calculation instead of beam line calculation. If `true`, `ProtonFieldVolumeDoseCapability` outputs the field verification reference point in addition to the dose in the final output.

Algorithm Property	Default value	Description
`SetCanCalculateVMAT`	false	If `true`, defines that the algorithm can calculate the dose to VMAT (dose dynamic arc) fields. Only meaningful for photon dose calculation algorithms.
`SetCanDoMonteCarloParallelization`	false	Indicates whether the algorithm is capable of Monte Carlo parallelization. If `true`, Eclipse can use several servants for calculating the dose for one field. Each servant gets a unique subfield number in the input. In this case, the algorithm only calculates the dose corresponding to the referenced subfield (that is, a smaller number of particle histories based on the subfield count).
`SetCanHandleNonStandardFluenceResolution`	false	Indicates whether the algorithm can handle block fluence matrices that have a resolution different from the default 2.5 millimeters.
`SetCanHandlePatientSupportDevice`	false	Indicates whether the algorithm takes patient support devices into account when calculating. The devices are sent to the calculation as structures with assigned densities. The algorithm should burn the structures into the 3D CT image it uses. This is very much similar to how bolus are accounted for.
`SetCanProtonOptimizeWithoutLowerConstraints`	true	Determines whether the proton optimization can be started without any explicit user defined lower constraints.
`SetDoesNotNeedPointCloudModel`	false	Determines whether the optimization algorithm does not use the point cloud calculated by Eclipse client. The point cloud is a representation of the volumetric structures. This representation is used by most Varian proprietary algorithms.
`SetDoesNotRequireBeamData`	false	Indicates whether the algorithm does not use beam data. By default this is false, meaning that the algorithm requires beam data.
`SetDoNotBurnAssignedCTValues`	false	See the description of `GetStructureSet`on page 87.
`SetMaxCalculationGridSizeX`	256	Sets the maximum calculation grid X-size (in voxels). Used in the Eclipse client to control the maximum size of the calculation volume.
`SetMaxCalculationGridSizeY`	256	Sets the maximum calculation grid Y-size (in voxels). Used in the Eclipse client to control the maximum size of the calculation volume.
`SetMaxCalculationGridSizeZ`	1000	Sets the maximum calculation grid Z-size (in voxels). Used in the Eclipse client to control the maximum size of the calculation volume.
`SetMaxDensityImageSizeX`	256	Sets the maximum X-size of the volume image. If the X-size of the volume image is larger than the value specified by this property, Eclipse will resample the image in the X-direction to this size.

Algorithm Property	**Default value**	**Description**
`SetMaxDensityImageSizeY`	256	Sets the maximum Y-size of the volume image. If the Y-size of the volume image is larger than the value specified by this property, Eclipse will resample the image in the Y-direction to this size.
`SetMaxDensityImageSizeZ`	1000	Sets the maximum Z-size of the volume image. If the Z-size of the volume image is larger than the value specified by this property, Eclipse will resample the image in the Z-direction to this size.
`SetNeedsAllStructures`	false	If true, the input structure set contains all structures. If false, the structure set contains body, PTV, bolus, couch structures, and structures that have a CT value assigned.
`SetNeedsSegmentModel`	true	Indicates whether the algorithm needs segment model.
`SetPhysicalMaterialTable`	(none)	Defines the physical material table that the algorithm supports. The algorithm can define the maximum of one physical material table that it supports. Each physical material definition must contain an ID, minimum/maximum/default mass density (g/cm^3), and material display name translations for the supported languages. The physical material table ID must be unique. The physical material table must also meet the following conditions: ▪ No mass density may belong to the range of three or more materials that can be automatically assigned. ▪ No materials that can be automatically assigned may have mass density range that entirely overlaps with that of another material that can be automatically assigned. ▪ The density range for materials that can be automatically assigned must be continuous, starting from the density 0.0012 g/cm3 or lower. ▪ The material table may not contain two materials with the same ID. ▪ The default density of any of the materials may not exceed the maximum density, or may not be less than the minimum density. ▪ The physical material table may not contain default, minimum or maximum densities with negative values. Before the physical material table is ready to be used in Eclipse, it must be imported into the ARIA database (see Chapter 4, “Installing and Uninstalling a Custom Algorithm”). See also: GetPhysicalMaterialTable request in “Table 5 Requests and Data Returned” on page 79.
`SetProtonOptimizationCapabilityUsesDDCCacheData`	false	Indicates whether the proton optimization capability in the servant needs proton DDC cache data.

Algorithm Property	Default value	Description
`SetProtonOptimizationSupportsNormalTissueConstraint`	false	Determines whether the proton optimization algorithm supports the normal tissue constraint. If it is supported, the corresponding GUI elements are enabled in the optimization dialog.
`SetSupportedMachineModels`	(empty)	Semicolon separated list of treatment machine model IDs that the algorithm servant supports.
`SetSupportedMLCModels`	(empty)	Semicolon separated list of MLC model IDs that the algorithm servant supports.

Chapter 11 Resource Usage Estimate Expression Variable Reference

Table 8: Variables available for each expression type and capability

Variable	HostRequirementsExpression	ExpectedProcessorUsageExpression	ExpectedMemoryUsageInMBExpression	ElectronFieldVolumeDoseCapability	PhotonOptimizationCapability	PhotonLMCCBSFCapability	PhotonLMCMonitorUnitsCapability	PhotonOptimalFluenceToLeafMotionsCapability	PhotonFluenceCapability	PhotonFieldVolumeDoseCapability	PhotonPlanVolumeDoseCapability	ProtonCompensatorCreationCapability	ProtonCompensatorConversionCapability	ProtonOptimizationCapability	ProtonFieldVolumeDoseCapability
HostName	■			■	■	■	■	■	■	■	■	■	■	■	■
ProcessorCount	■	■		■	■	■	■	■	■	■	■	■	■	■	■
CalculationAreaXSizeInMM			■	■	■					■	■	■	■	■	■
CalculationAreaYSizeInMM			■	■	■					■	■	■	■	■	■
CalculationAreaZSizeInMM			■	■	■					■	■	■	■	■	■
CalculationGridSize			■	■		(■)	(■)			■	■	■	■	■	■
FieldLengthInMM			■	■			■			■	■	■	■	■	■
FieldTechnique			■	■			■			■	■	■	■	■	■
FieldXSizeInMM			■	■			■			■	■	■	■	■	■
FieldYSizeInMM			■	■			■			■	■	■	■	■	■
ImageXSizeInMM			■	■	■					■	■	■	■	■	■
ImageXSizeInPixels			■	■	■					■	■	■	■	■	■
ImageYSizeInMM			■	■	■					■	■	■	■	■	■
ImageYSizeInPixels			■	■	■					■	■	■	■	■	■
NumberOfFields			■	■	■	■	■			■	■	■	■	■	■
NumberOfPoints			■		■									■	
NumberOfSlices			■	■	■					■	■	■	■	■	■
SliceIntervalInMM			■	■	■					■	■	■	■	■	■

Table 9 Variables that can be used in resource usage estimate expressions

Variable	Description
`HostName`	Computer name.
`ProcessorCount`	Number of processors (physical cores).
`CalculationAreaXSizeInMM`	Calculation volume X-size in millimeters.
`CalculationAreaYSizeInMM`	Calculation volume Y-size in millimeters.
`CalculationAreaZSizeInMM`	Calculation volume Z-size in millimeters.
`CalculationGridSize`	Calculation grid cell size in centimeters. `CalculationGridSize` is only defined if the calculation options of the algorithms contain the attribute `CalculationGridSizeInCM`.
`FieldLengthInMM`	Field length from body entry to body exit on the field central axis.
`FieldTechnique`	`"ARC"`, `"STATIC"`, `"SRS_STATIC"`, `"SRS_ARC"` This variable can be used only in if-then-else expressions.
`FieldXSizeInMM`	Field X-size in millimeters.
`FieldYSizeInMM`	Field Y-size in millimeters.
`ImageXSizeInMM`	Image X-size in millimeters.
`ImageXSizeInPixels`	Image X-size in pixels.
`ImageYSizeInMM`	Image Y-size in millimeters.
`ImageYSizeInPixels`	Image Y-size in pixels.
`NumberOfFields`	Number of fields in the plan.
`NumberOfPoints`	Number of points in the point cloud.
`NumberOfSlices`	Number of slices in the image.
`SliceIntervalInMM`	Slice interval in millimeters.

Chapter 12 Troubleshooting

Q: Compilation shows an error about incompatible Visual Studio version.

A: Check that the Visual Studio version you are using exactly matches the version specified in "System Requirements" on page 13. Eclipse Algorithm API is strict about the correct Visual Studio version, due to the possible template/runtime library (and therefore, for instance, memory layout) differences and the resulting issues.

Q: Visual Studio linker shows warnings about EAAPI-related .pdb files.

A: EAAPI has been compiled such that it generates the .pdb files. As those .pdb files are not included in the package, Visual Studio's linker shows the warnings. There is no option in Visual Studio 2013 to turn that specific warning off. So those warnings are to be expected.

Q: Eclipse complains about missing calculation options for my algorithm, or the `GetCalculationOptionsXmlString` request fails.

A: Check that you have copied your algorithm-version-specific Option Schema XSD and Default Options XML files to the `<DCFDirectory>\client\Options ([API])` directory.

Q: When trying to use my custom algorithm in Eclipse, I get the following error message: "The custom algorithm uses an unsupported EAAPI version. Only Eclipse Algorithm API 13.0.X is supported."

A: The algorithm has been compiled with an earlier version of EAAPI. Recompile your algorithm with the version 13.0 header files and libraries.

Q: When trying to use my custom algorithm in Eclipse, I get the following error message: "Custom (EAAPI) algorithms can only be used with research database."

A.Contact Varian Customer Support to have your Eclipse research system database configured for research use. This should be performed by your Varian Customer Support Representative.

Q: My servant does not start/execute.

A: Check that:

- You have `VCServant.dll` (for Release configuration) or `VCServantd.dll` (for Debug configuration) in your servant executable directory.
- The directory of the servant executable contains all the needed binaries (including runtime libraries) as well as resources.
- The servant is properly registered to `VCConfiguration.xml`.
- The servant is properly registered to `InstalledAlgorithms.xml`.
- You have restarted Eclipse after making changes to the DCF configuration.

Specifically, if Eclipse states that it cannot find a service, and you have recently updated your servant executable, check that you have updated the registration in `VCConfiguration.xml` and `InstalledAlgorithms.xml` as well. Also, remember to restart Eclipse after modifying the DCF configuration.

Q: My servant crashes.

A: See Chapter 6 "Starting and Debugging Custom Algorithms" on page 49 for instructions on how to turn the debug output on from Eclipse. See also "Creating a Visual Studio Project and a Main Function for the Servant" on page 23 and ensure that you have debug iterators and checked iterators turned off in your Visual Studio projects. Having them on is not supported and is expected to cause issues.

Q: My capability cannot resolve the beam data directory path.

A: Check that

- You have configured the beam data directories in Beam Configuration for the calculation model of your algorithm.
- Your capability removes the trailing padding from the treatment machine ID, because it can contain trailing spaces. For removing the padding, you can use the `EAAPI::Utils::RemoveTrailingPadding` method.

Q: I get the following error message: "Algorithm error: Invalid exit (the capability is currently in '*XXX*' state)."

A: The capability did not exit in a valid way. If the calculation was successful, the valid way to exit is to take the following steps, in the following order:

1. Call `requester.PutCalculationStartNotification`
2. Call `requester.PutCalculationFinishedNotification` and
3. Send the final output

If your algorithm encounters an error, use `requester.PutErrorFlag` to signal the error to Eclipse.