Using the strategy pattern to simplify ITK

Release 1.00

Dan Mueller[1]

January 6, 2011

[1]Philips Healthcare, Best, Netherlands

Abstract

ITK is becoming increasingly complex as it continues to grow and evolve. The SimpleITK initiative aims to address this concern by providing an easy-access layer around ITK for non-C++ expert developers. This article proposes an alternate (perhaps complementary) approach utilizing the strategy pattern. The software design pattern known as "strategy" allows an algorithm to be selected from a family of algorithms on-the-fly at runtime. Because ITK consists of many such algorithm families, this particular design pattern is quite interesting. This article describes a number of algorithm families which have been identified; so far thirteen families have been implemented, encapsulating nearly 130 filters/objects. Complete source code and examples are provided to demonstrate the concept.

Latest version available at the Insight Journal [http://hdl.handle.net/10380/3248]

Contents

1 Introduction

ITK has grown considerably over the course of its 10 year history. At the time of writing, the toolkit contains roughly 1500 classes within 3000 source code files (counting only the Code directory). This amount of code is daunting for many prospective users. This article proposes the use of the software design pattern known as "strategy" to help address this issue. A clear description of the strategy pattern is given in [1]. As depicted in the UML class diagram in Figure 1, the strategy pattern represents a family of algorithms. This family is encapsulated behind a common interface, with the specific algorithm or implementation (known as the concrete strategy) chosen at run-time. Additionally an operational context can be provided, but this concept has not be utilized in the described approach.

The original intent of the strategy pattern was that different implementations provide the *same* behavior; that is, each concrete strategy must implement the same common interface. Within this article, that intention has been stretched: implementations need only have a *similar* interface, with some additional methods on the concrete strategies. This was done to allow a larger number of filters/objects to be grouped behind the strategy interface.

The following section describes the algorithms families which have so far been identified and implemented. Next some source code examples are given to demonstrate intended usage. Finally various limitations and future work is discussed.

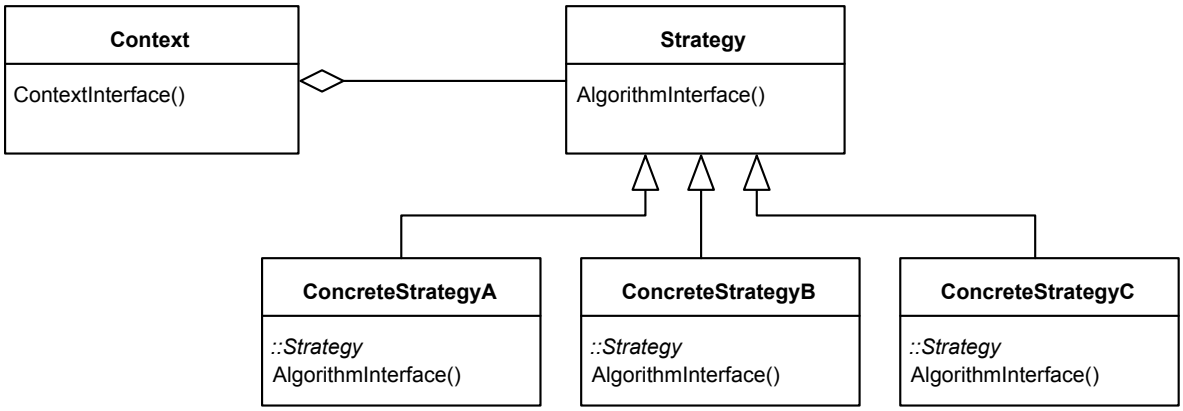

Figure 1: UML class diagram of strategy pattern.

2 Description

To date, thirteen algorithm families have been identified and implemented. These are listed in the following tables:

UnaryPixelMathImageFilter	Abs, Acos, Asin, Atan BoundedReciprocal Cos, Exp, ExpNegative InvertIntensity Log10, Log Not, Sigmoid, Sin Sqrt, Square, Tan
BinaryPixelMathImageFilter	Add, Subtract, Multiply, Divide WeightedAdd AbsoluteValueDifference BinaryMagnitude ConstrainedValueAddition ConstrainedValueDifference Mask, MaskNegated Maximum, Minimum SquaredDifference
DistanceMapImageFilter	Danielsson SignedDanielsson SignedMaurer
GradientImageFilter	FiniteDifference RecursiveGaussian
GradientMagnitudeImageFilter	FiniteDifference RecursiveGaussian Morphological Laplacian LaplacianRecursiveGaussian
MorphologyImageFilter	Dilate, Erode, Close, Open BinaryDilate, BinaryErode, BinaryClose, BinaryOpen CloseByReconstruction, OpenByReconstruction BlackTopHat WhiteTopHat GeodesicDilate, GeodesicErode DilateByReconstruction, ErodeByReconstruction AreaClose, AreaOpen ConnectedClose, ConnectedOpen HConcave, HConvex, HMaxima, HMinima RegionalMaxima, RegionalMinima

ProjectionImageFilter	Binary
	BinaryThreshold
	Maximum
	Minimum
	Sum
	Mean
	StandardDeviation
	Median
SmoothImageFilter	Bilateral
	CurvatureFlow
	DiscreteGaussian
	RecursiveGaussian
	Median
ThresholdImageFilter	Binary
	Single
	Double
	Otsu
	KappaSigma
InterpolateImageFunction	NearestNeighbor
	Linear
	BSpline
	BlackmanWindowedSinc
	CosineWindowedSinc
	HammingWindowedSinc
	LanczosWindowedSinc
	WelchWindowedSinc
ImageToImageMetric	CompareHistogram
	CorrelationCoefficientHistogram
	GradientDifference
	KappaStatistic
	KullbackLeiblerCompareHistogram
	MatchCardinality
	MattesMutualInformation
	MeanReciprocalSquareDifference
	MeanSquaresHistogram
	MeanSquares
	MutualInformationHistogram
	MutualInformation
	NormalizedCorrelation
	NormalizedMutualInformationHistogram

Optimizer	Amoeba
	ConjugateGradient
	FRPR
	GradientDescent
	LBFGSB
	LBFGS
	LevenbergMarquardt
	OnePlusOneEvolutionary
	Powell
	RegularStepGradientDescent
	VersorRigid3DTransform
	VersorTransform
Transform	Translation
	Scale
	Euler2D
	Euler3D
	Versor
	VersorRigid3D
	Similarity2D
	Similarity3D
	Affine
	BSpline

3 Examples

The best way to understand the potential use is through source code examples. This section gives three examples, however the accompanying source code contains test code for all of the implemented strategies.

3.1 Registration

```
// Read strategies from command line
itk::TransformStrategy TransformStrategy =
  static_cast<itk::TransformStrategy>( atoi(argv[arg++]) );
itk::OptimizerStrategy OptimizerStrategy =
  static_cast<itk::OptimizerStrategy>( atoi(argv[arg++]) );
itk::ImageToImageMetricStrategy MetricStrategy =
  static_cast<itk::ImageToImageMetricStrategy>( atoi(argv[arg++]) );
itk::InterpolateImageFunctionStrategy InterpolatorStrategy =
  static_cast<itk::InterpolateImageFunctionStrategy>( atoi(argv[arg++]) );

// Helpful typedefs
typedef itk::SimpleTransform<CoordRepType,Dimension,Dimension> TransformType;
typedef itk::SimpleOptimizer OptimizerType;
typedef itk::SimpleImageToImageMetric<ImageType,ImageType> MetricType;
typedef itk::SimpleInterpolateImageFunction<ImageType,CoordRepType> InterpolatorType;
typedef itk::ImageRegistrationMethod<ImageType,ImageType> RegistrationType;
typedef itk::ResampleImageFilter<ImageType,ImageType,CoordRepType> ResampleType;

// Transform
typename TransformType::Pointer transform = TransformType::New();
transform->SetStrategy(TransformStrategy);
transform->SetParameters(initial);

// Interpolator
typename InterpolatorType::Pointer interpolator = InterpolatorType::New();
interpolator->SetStrategy(InterpolatorStrategy);

// Optimizer
typename OptimizerType::Pointer optimizer = OptimizerType::New();
optimizer->SetStrategy(OptimizerStrategy);
optimizer->SetScales(scales);

// Metric
typename MetricType::Pointer metric = MetricType::New();
metric->SetStrategy(MetricStrategy);

// Registration
typename RegistrationType::Pointer registration = RegistrationType::New();
registration->SetFixedImage(fixedImage);
registration->SetMovingImage(movingImage);
registration->SetOptimizer(optimizer);
registration->SetMetric(metric);
registration->SetTransform(transform);
registration->SetInterpolator(interpolator);
registration->SetInitialTransformParameters(initial);
registration->Initialize();
registration->StartRegistration();
```

3.2 Binary Pixel Math

```
    // Read strategy from command line
  itk::BinaryPixelMathStrategy Strategy =
    static_cast<itk::BinaryPixelMathStrategy>( atoi(argv[arg++]) );
  std::cout << "WeightedAddAlpha=" << WeightedAddAlpha << std::endl;

  // Helpful typedefs
  typedef itk::Image<PixelType, Dimension> ImageType;
  typedef itk::SimpleBinaryPixelMathImageFilter<ImageType,ImageType> BinaryPixelMathType;

  // Filter
  typename BinaryPixelMathType::Pointer filter = BinaryPixelMathType::New();
  filter->SetStrategy(Strategy);
  filter->SetInput1(input1);
  filter->SetInput2(input2);
  filter->Update();
```

3.3 Threshold

```

  // Read strategy from command line
  itk::ThresholdStrategy Strategy =
    static_cast<itk::ThresholdStrategy>( atoi(argv[arg++]) );

  // Read other parameters
  TPixel Outside = (argc > arg) ? (TPixel)atof(argv[arg++]) : (TPixel)0;
  TPixel Inside = (argc > arg) ? (TPixel)atof(argv[arg++]) : (TPixel)0;
  TPixel Lower = (argc > arg) ? (TPixel)atof(argv[arg++]) : (TPixel)0;
  TPixel Upper = (argc > arg) ? (TPixel)atof(argv[arg++]) : (TPixel)0;

  // Helpful typedefs
  typedef itk::Image<PixelType, Dimension> ImageType;
  typedef itk::SimpleThresholdImageFilter<ImageType> ThresholdType;

  // Filter
  typename ThresholdType::Pointer filter = ThresholdType::New();
  filter->SetStrategy(Strategy);
  filter->SetInput(input);
  filter->SetOutsideValue(Outside);
  if (Strategy != itk::ThresholdStrategySingle)
    {
    filter->SetInsideValue(Inside);
    }
  if ((Strategy == itk::ThresholdStrategySingle) ||
      (Strategy == itk::ThresholdStrategyBinary))
    {
    filter->SetLowerThreshold(Lower);
    filter->SetUpperThreshold(Upper);
    }
  filter->Update();
```

4 Conclusion

This article and accompanying source code describes the use of the strategy software pattern to help simplify ITK. A number of similar filters or objects are encapsulated behind a common interface, with the ability to select the desired algorithm at runtime. Together with SimpleITK and/or WrapITK the given approach has the ability to dramatically reduce the number of classes (new) ITK developers need know about.

The "umbrella" strategy objects remain templated (typically over the image type), but together with WrapITK this limitation can be alleviated. Also, in order to group larger numbers of filters/objects, the strategy interface is not always common for all algorithms. Compounded with the use of templates, this can sometimes result in ugly switch statements[1] which ironically the strategy pattern is intended to prevent [1, pg. 317]. This too can be alleviated by making all algorithms derive from the same subclass[2]. (It should be noted these ugly switch statements are not needed in user code, only within the strategy filter or object).

Obviously there is still work to be done. Other algorithm families can be identified and encapsulated; some examples include NaryPixelMathImageFilter, RegionGrowingImageFilter, and LevelSetImageFilter. Of the existing algorithm families, a number of algorithm specific parameters/methods still need to be exposed at the common interface level; for example many of the optimizers do not have their specific parameters exposed.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1977. 1, 4

[1] See itkSimpleMorphologyImageFilter.h for an ugly example.
[2] See itkSimpleProjectionImageFilter.h for a good example.