Object-oriented software design in semiautomatic building extraction

E. Gülch and H. Müller

Institute for Photogrammetry, University of Bonn
D-53115 Bonn, Germany

ABSTRACT

Developing a system for semiautomatic building acquisition is a complex process, that requires constant integration and updating of software modules and user interfaces. To facilitate these processes we apply an object-oriented design not only for the data but also for the software involved. We use the Unified Modeling Language (UML) to describe the object-oriented modeling of the system in different levels of detail. We can distinguish between use cases from the users point of view, that represent a sequence of actions, yielding in an observable result and the use cases for the programmers, who can use the system as a class library to integrate the acquisition modules in their own software. The structure of the system is based on the Model-View-Controller (MVC) design pattern. An example from the integration of automated texture extraction for the visualization of results demonstrate the feasibility of this approach.

Keywords: Unified Modeling Language, Texture, Constructive Solid Geometry, User Interface, Model-View-Controller, Design Pattern

1. INTRODUCTION

A system for semiautomatic building acquisition from digital imagery is being built up from various interactive and automated modules, that are already existing or that are under development.[1–3] Such a system has to be extremely flexible and puts high requirements on the software design. It should be able to:

- solve tasks, defined by users from many different application fields,
- allow the introduction of automated modules for certain subtasks without redesigning the whole system,
- apply the software to a great variety of sensor data, ranging from mono-, stereo- to multiple imagery (close-range, aerial or satellite) to DEM and GIS data of various scale.

This means the involved software must fulfill requirements like: reuse, extendibility, portability and maintenance. To reuse software in topographic applications is of great importance, due to the high variability of sensor data and tasks. New tasks and new sensors require extensions by new modules, mainly automation of certain sub-tasks. An example for that is the introduction of new sensor types with different projection, like e.g. SPOT imagery. The portability of the software to new platforms must be guaranteed due to the high rate of revision changes in the computer industry. For maintenance it is advantageous to build on data encapsulation. This eases testing and changing of current revisions of the software.

The object-oriented software design offers advantages for such a constantly growing system, that is updated on weekly or monthly rates. We apply object-orientation not only to classical data structures like image data or 3D-vector data, but to the whole system design by combining algorithms *and* data to objects. The task is, however, not trivial due to the fact, that the complexity and connectivity of existing software is very high.

Other author information: (Send correspondence to E.G.)
E.G.: Email: ebs@ipb.uni-bonn.de; Telephone: +49-228-73-2904; Fax: +49-228-73-2712;
H.M.: Email: hardo@ipb.uni-bonn.de; Telephone: +49-228-73-2721; Fax: +49-228-73-2712.
This research is supported by DARA-GmbH under Contract 50 TT 9536.
In Integrating Photogrammetric Techniques with Scene Analysis and Machine Vision III, SPIE Publications 3072-15, Orlando, USA, 21-25 April 1997.

We use the Unified Modeling Language (UML) to describe the object oriented modeling of the system in different levels of detail (cf. Section 2). We distinguish between the use cases from the user's and programmer's point of view and the static structure of the system (cf. Section 3). To illustrate our approach we give examples from the object-oriented integration of automated texture extraction for visualization and control of the extracted 3D buildings (cf. Section 4).

2. OBJECT-ORIENTATION AND THE UNIFIED MODELING LANGUAGE

This section gives a short introduction into object-orientation, by defining some of the major expressions relevant here, and describes the Unified Modeling Language (UML), which we use in our notation.

2.1. Object-orientation

Definitions of "Object-oriented programming" and "Object-oriented design" are described in Booch(1994)[4]:

> "Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships."

> "Object-oriented design is a method of design comparsing the process of object-oriented decomposition and a notation for depicting the logical and physical as well as static and dynamic models of the system under design."

An object-oriented system consists of a set of objects, which are defined by different *classes*. The integral part of a class are methods and attributes. The attributes describe the data of an object, and the methods describe its behavior. To prevent an illegal access to internal attributes or methods of a class, an object-oriented language offers different levels of encapsulation. This means that e.g. a C++ class is able to contain *public*, *protected* and *private* elements.

The classes of an object-oriented system are associated by different types. The essential ones are (cf. also Figure 1):

- **Inheritance**

  Class Subclass contains all elements of class Superclass and some additional features. In this case it is advantageous for class Subclass to inherit elements from class Superclass. Another aspect of inheritance is the association between an abstract class and its implementation. An abstract class contains only the signature of its methods, but not the definition. Only objects of inherited classes, which have the abstract methods implemented can be instantiated.

- **Aggregation**

  Some attributes of class Whole are instances of class Part. In other words: Whole contains Part.

- **Uni-Directional Association**

  Class Client contains a reference to an object of class Supplier, but class Supplier does not know anything about class Client.

- **Bi-Directional Association**

  Class A and B contain references to each other.

- **Dependency**

  The methods of class Client use methods or attributes of class Supplier.

- **Template Instantiation**

  A class Template can be parameterized by the type parameter T. For example the class Template<float> is an instantiation of Template with the type parameter float.

The design process of an object-oriented system starts by modeling a set of classes and their associations. A special notation of object-oriented modeling will ease the work of the designer.

2.2. The Unified Modeling Language

Neither a natural language nor a programming language is able to describe the modeling of a complex system in an easy way. For that reason G. Booch, I. Jacobson and J. Rumbaugh designed the Unified Modeling Language.[5] The UML notation provides a variety of diagrams for different purposes, like dynamic or static system views.

Figure 1 gives an overview of the UML notation, which is used in the following diagrams, that describe our system (c.f. Figures 2 - 7). The association types as described above are elements of *Static Structure*- or *Class Diagrams*, that show the static structure of the model. These diagrams contain the essential classes, *packages* and their *associations* (e.g. Fig. 5 and 6). A *Use Case Diagram* shows the relationship among *actors* and use cases within a system (cf. Fig. 2). The main actors in our system are the user (operator) and the programmer. A *Sequence Diagram* is attached to an operation or an use case. It describes the *objects* participating in the interaction by their *life-lines* and their exchanged *messages* during a time sequence. The process of texture extraction (cf. Fig. 7) is e.g. described by a *Sequence Diagram*.

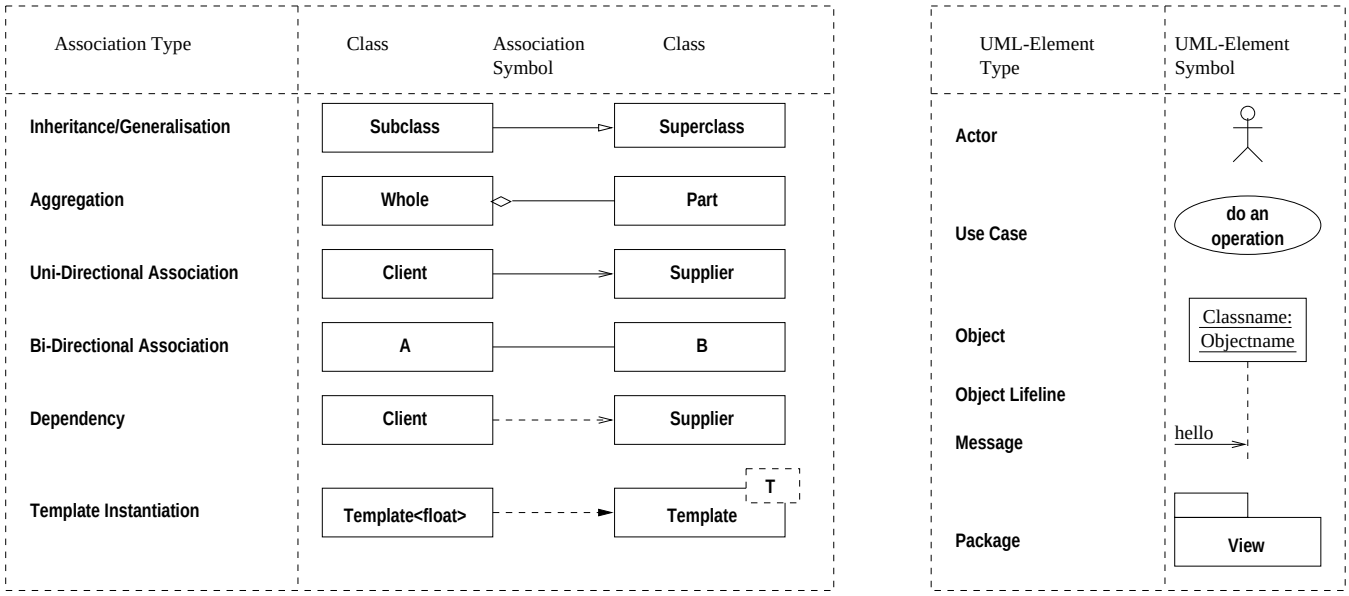


Figure 1. UML-Notation of Association Types (left Box) and other UML-elements(right box).

3. SYSTEM VIEWS

The two major views of the system are the use cases on one hand side and the static structure on the other hand side. The use cases contain the user's and the programmer's point of view, the static structure contains the major objects of interest, the models and the images.

3.1. Use Cases

A use case represents a sequence of actions, which yield in an observable result.[5] We distinguish between two major groups, the Use Cases for the Operator (User) and the Use Cases for the Programmer. The essential use cases of the Semiautomatic Building Extraction System for both groups are shown in the use case diagram in Figure 2 as ellipses containing the names of the use cases.

3.1.1. Use Cases for the operator (user)

These use cases contain all actions by the operator, the invocation of the automated modules for model adaptation, height measurement, visualization, texture extraction etc.

The operator can freely *navigate* through the images and change between different resolution steps (cf. Fig. 2). The system can handle completely digitized aerial images.

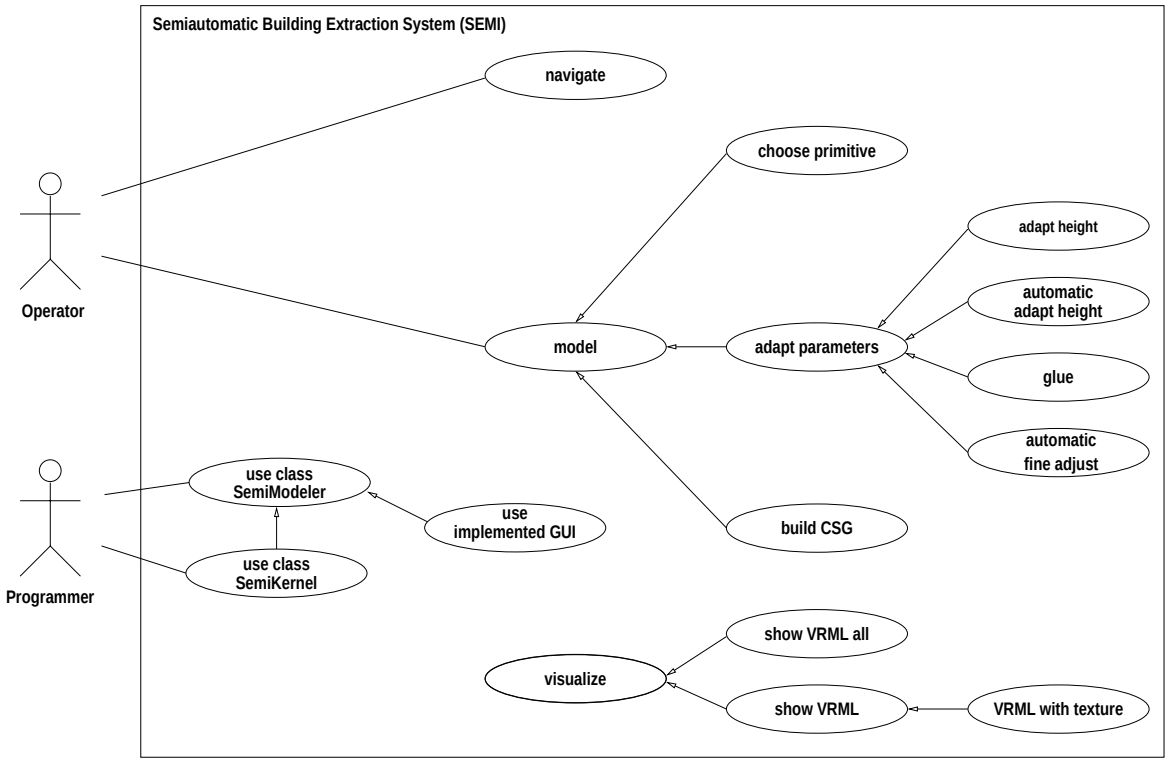


Figure 2. Use Case Diagram of the Semiautomatic Building Extraction System.

After navigating to the objects of interest, the operator starts to *model* the objects. First, he has to *choose a 3D-primitive*. A wire frame model is projected into the image. The operator is now able to *adapt the parameters* of the model. This can be done interactively by clicking at one or two points of the model and moving the pointer until the parameters are adjusted.[1,3] If the operator *builds a CSG-tree* by combining primitives, he can use a gluing tool for a precise "docking" of the primitives. This tool is automated and matches new primitives to already existing ones, based on face-to-face matching or parameter to parameter matching.

After having adjusted the form and pose in one image, there is only one undetermined parameter left: the scale or the absolute height. To adapt the absolute height in 3D, the operator can choose between several options:

1. To *measure* a homologous point by identifying one point of the model (usually on top of the roof) and measuring it precisely in the other image(s) with the cursor. This option is the conventional one, and most time consuming. No stereo-viewing is applied.

2. To *drag a slider* that moves the defined model in one image, along the rays defined by the model points and the projection center until the model fits in the other image(s). In this way, not only a single point is measured, but all visible model lines are used to adjust the height, by adapting them in the other image(s). This method is faster and more reliable.

3. If the image data is of good quality, the operator can select an *automated height adapting* instead of dragging a slider. The adapted model in the first image (fig. 3) is automatically fit to previously fully automatically extracted line segments in the other image(s) as shown in fig. 4. Robust pose clustering techniques[6] are used to determine the height in 3D.

An *automatic fine adjustment*[7] by a robust spatial resection, using all line segments in all images provides an optimal fit of the selected model to the image data.

Figure 3. A 3D-primitive (saddleback-roof building) is pose and form adapted in the left image. The height is not yet measured, as shown in the right image.

Figure 4. A 3D-primitive (thick black lines) with measured height. Some of the automatically extracted line segments (thick white lines) have been used for a final fine tuning of the parameters.

The result of the model adaptation and height measurement are 3D-coordinates of the building stored in a CSG-tree. Those can be exported to DXF-file-format as input for analysis in standard CAD systems.

A *visualization* of the results is done by converting the building data to the Virtual Reality Modeling Language (VRML) and exporting it to a VRML-browser. To get an overview and to check for completeness, the operator can select the use case '*show VRML all*' to see all acquired models in the browser. Alternatively he can select '*show VRML*', and the model is shown, he is currently working on. Optionally it is possible to visualize a VRML-model with automatically *extracted texture*, to check for consistency or to derive a photorealistic view for animation purposes (cf. also Section 4).

3.1.2. Use cases for the programmer

A programmer can use the system as a class library to integrate the acquisition modules in other software.

For the integration an ad-hoc solution to call external software in a C/C++ program could be done by the following function call:

```
system("external_program");
```

But this way is not very flexible. The external program is started while the calling program is stopped, until the external program has finished. This feature is very disadvantageous for programs involving interaction. There is no easy way to communicate with the calling program.

The object orientation of our Semiautomatic Building Extraction System (Semi) enables another way of using the system in the programmer's own software, e.g. as a building acquisition tool for a wavefront propagation package of a mobile phone company. It offers an interface of a class library. A programmer can use this class library in two different levels:

1. If the programmer applies the same Graphical User Interface (GUI) library as implemented in our system, he is able to instantiate an object of class SemiModeler by the following C++ instruction:

   ```
   SemiModeler modeler;
   ```

   The object modeler is inherited from a widget of the implemented GUI library, and the programmer can use it to communicate with his custom widgets. It gives a reference to the GUI-independent kernel of the system by the following method call:

   ```
   SemiKernel& semi = modeler.Kernel();
   ```

2. If the programmer does not want to use the implemented GUI, he can use the system with his own GUI by creating the kernel directly:

   ```
   SemiKernel semi;
   ```

   The programmer can create his own widgets, which communicate by the class interface of SemiKernel with the object semi. This makes it possible to use the system within another system with its own graphical user interface.

3.2. Static structure

The static structure of the Semiautomatic Building Extraction System contains the major objects of interest, the models and the images. It is based on the Model-View-Controller (MVC)[8] design pattern.

The static structure is shown in Figure 5. Significant classes, such as image classes and 3D-model classes like the TextureModel class are stored in the Model package. The Model package is independent of the GUI, and the contained classes can be reused within another context.

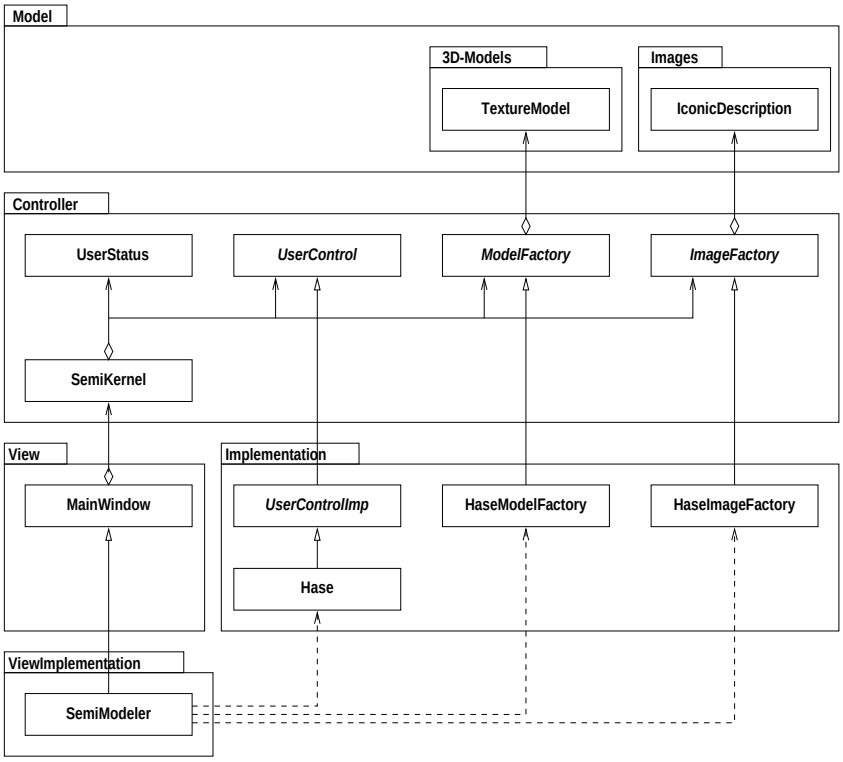


Figure 5. The Static Structure Diagram of the semiautomatic building extraction system.

The View and ViewImplementation packages contain the classes of graphical representation such as Windows and Widgets. This packages are designed to be easy exchangeable, because the graphical elements are highly platform dependent, and there exists no standard of GUI classes.

The communication between the classes in the View package and the classes in the Model package is done by the classes of the Controller package. This package contains additionally two factory-classes: ImageFactory and ModelFactory. These classes are designed based on the Abstract Factory pattern.[8] An Abstract Factory is a class, which enables to create objects without naming its real type. Their clients are able to access objects of abstract classes by remaining independent of their implementation.

The implementation of the abstract classes in the Model and Controller packages is encapsulated in the Implementation package. The classes of this package are not accessible for external objects. The code of the earlier system HASE+,[1] which is written in C, has been reused to implement these classes.

4. THE TEXTURE MODEL

We would like to illustrate the object-oriented design by the integration of an automated texture extraction module. Texture mapping on the faces of the extracted building primitives can be used for the final output, for visualization and animation purposes, as well as for an on-line control of the acquired building(s) for the operator.

4.1. Static model

Spatial objects with a texture representation need an appropriate description by a class interface. The class TextureModel represents such kind of objects in the scope of the Semiautomatic Building Extraction System. Figure 6 shows a class diagram with the relevant associations of TextureModel to other classes.

We assume, that the geometry of the acquired spatial objects can be described by vertices, edges and faces. This kind of description is called Vef-Graph.[9] Therefore the geometric description is represented by the methods of the class VefRepresentable. This class is a template class with a vector type as template parameter. It depends only on

containers of the Standard Template Library (STL)[10] and no other classes. For that reason it is easily possible to reuse this class, because the user does not need any more class libraries than the STL.

For the image representation we need a special image class, which is called TextureImage. This class inherits from VefRepresentable<PlaneVec> with PlaneVec as a 2D-vector type and from MatrixRepresentable with the image data type as template parameter. The purpose of this class is to offer an access to the image data as defined in the interface definition of MatrixRepresentable.

The class TextureRepresentable inherits from VefRepresentable and its methods describe an object with vertices, edges, faces and texture. It is a template class with the image data type as parameter. Special methods of TextureRepresentable enable texture access by returning objects of class TextureImage. Otherwise the class is designed in the same way as VefRepresentable. The class interface is able to represent texture from more than one image. The client can get the texture image of a special face, as well as all faces of a special texture image. The classes VefRepresentable and TextureRepresentable are abstract and contain no data. They can only be instantiated by a non abstract inherited class.

The relation between the 3D-faces and texture is defined in the class TextureFaceList, which represents a list of faces and their associations to the TextureImage objects. This class is also responsible for the visibility of the texture.

The class TextureModel inherits from TextureRepresentable and is consequently an implementation of VefRepresentable and TextureRepresentable. This class needs for its creation a reference to another object of class VefRepresentable<SpatialVec> as the 3D-model. The only template parameter of this class is the type of the image data.

4.2. Texture extraction process

The texture extraction process takes place in the method *addTexture*(cf. Figure 7) of the class TextureModel. The purpose of *addTexture* is to add one new oriented image of texture data to the model and to attach the texture to the faces by considering the visibility of the face. The following C++ code example shows the creation of a texture model, and the attachment of two images to it by using the method *addTexture*:

```
...
// Create a 3D-model which is able to contain texture data.
TextureModel texture_model(model_3d);

// Add texture data from two images to the model.
texture_model.addTexture(image1);
texture_model.addTexture(image2);
...
```

Texture extraction is done by creating an instance of class TextureModel and adding oriented image data to it. A step by step description of the method *addTexture* is given in the following and shown as a sequence diagram in Figure 7:

Step 1: Get projection from the oriented image.
Step 2: Create a 2D-model from the projection and the 3D-model.
Step 3: Get the projected points from the 2D-model.
Step 4: Extract an image patch defined by the bounding box of the projected points.
Step 5: Transform the projected points to the coordinates of the new image patch.
Step 6: Create a texture image from the image patch and the projected points.
Step 7: Connect the texture image to each face of the texture-face list by the following steps:
- (a) If the face is already connected with a texture image, compare the visibility of the new image with the visibility of the already connected image.
- (b) If the visibility of the new image is better or the face has not been connected to an image, connect the new image to the face and set the visibility of this face in the texture image `true`.
- (c) If the face is invisible in this image or the face is better visible in the already connected image, set the visibility of this face in the texture image `false`.

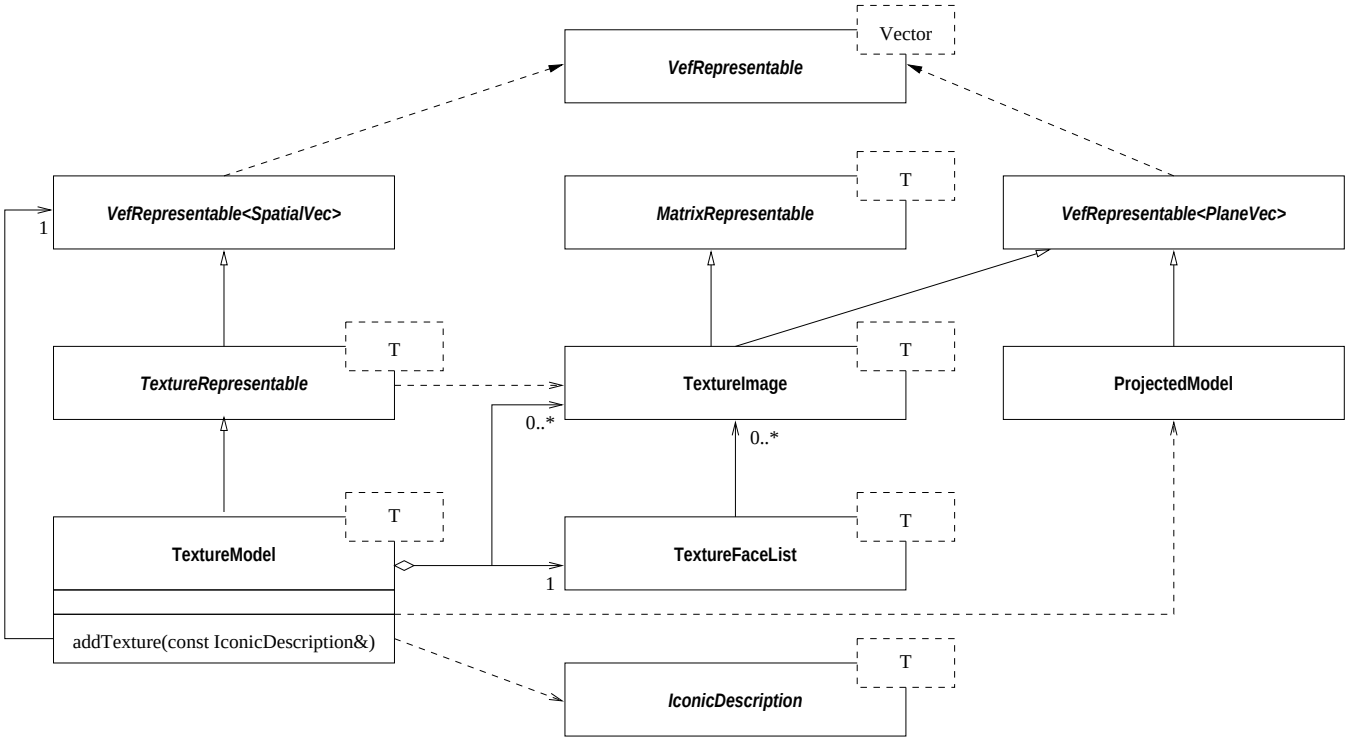


Figure 6. Static Structure Diagram of the texture model.

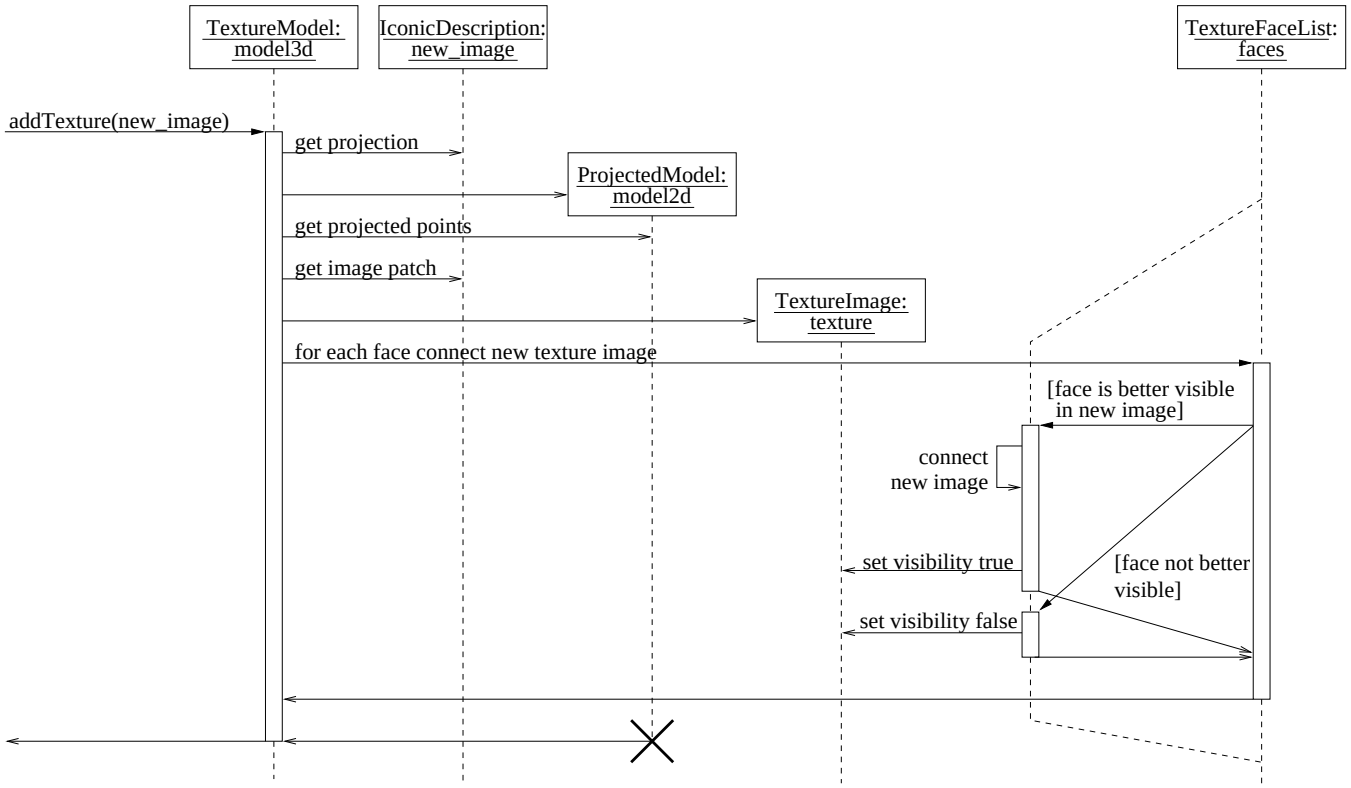
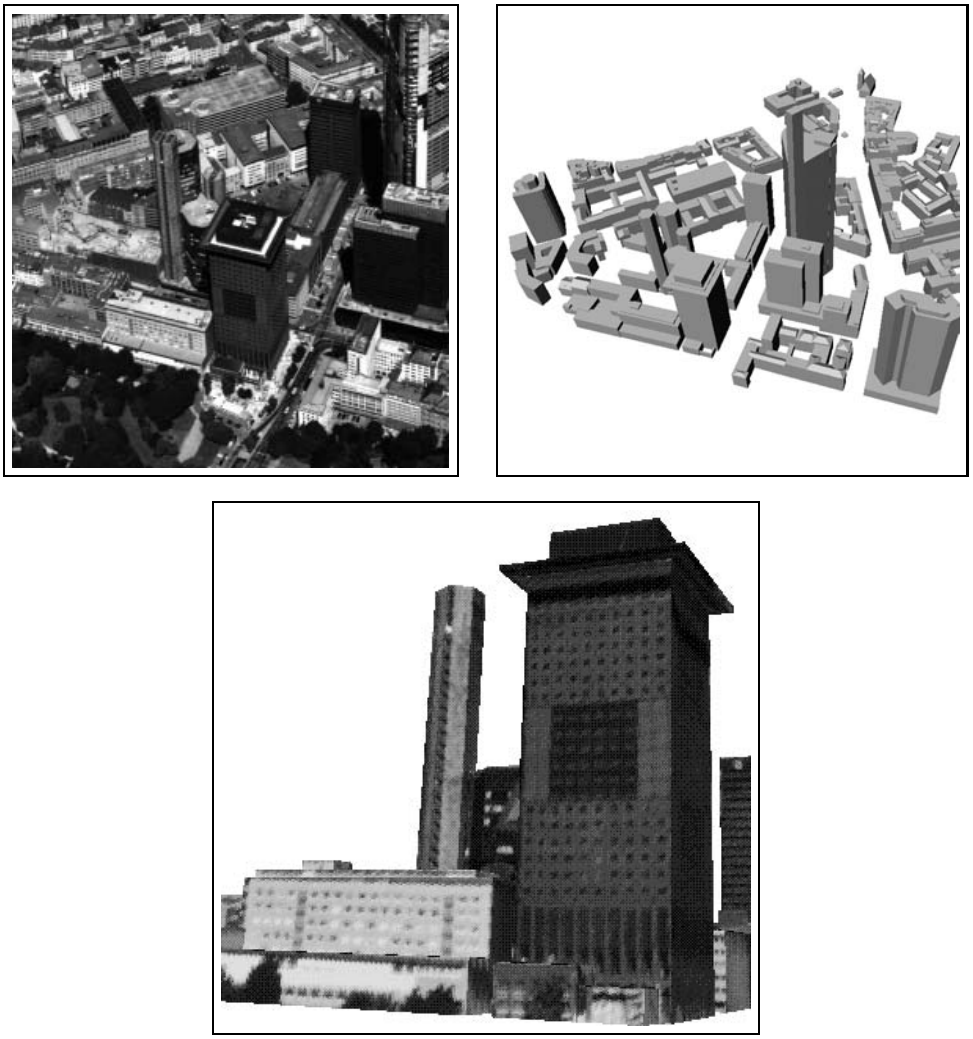


Figure 7. Sequence Diagram of the method *TextureModel::addTexture*.

4.3. Example

In a first example we have applied the automatic texture extraction to a 3D-building model from downtown Frankfurt/Main and a single aerial image (ground heights were given as well). Figure 8 shows the aerial image, the extracted 3D-building models from the semiautomatic system and the horizontal view of some buildings, whose faces have been automatically texture mapped using the steps described above. This example demonstrates the visualization potential of photorealistic views for all kinds of city planning purposes.

Aerial image 3D-building models



Horizontal view

Figure 8. Texture extraction from one aerial image of downtown Frankfurt/Main. Image data by courtesy of Stadtvermessungsamt Frankfurt.

In a second example we applied the described method *addTexture* to a complex 3D-building and *two* aerial images. In the upper row of Figure 9 the two image patches of the left and right image are given. Not all faces of the church are visible in both images, some are not visible at all. The lower left image shows the texture extraction result from the left image only. Invisible faces are marked black. The lower right image shows the texture extracted from both images. Some of the invisible faces from the left view, have now been mapped with texture from the right image.

Even windows and doors can be identified. By using texture mapping we do not need to model all geometric details, if not explicitly required, and still get highly realistic views of the objects.

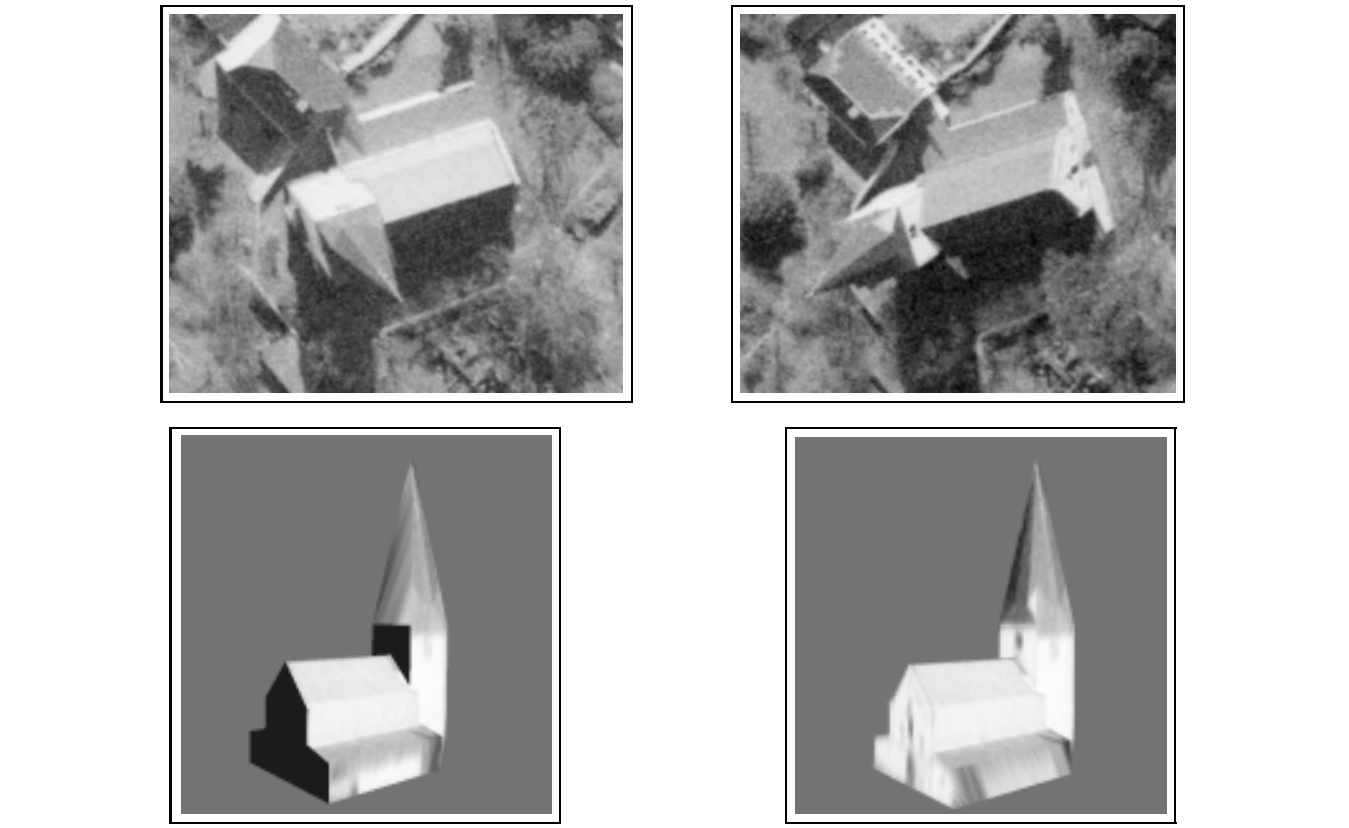


Figure 9. Automated texture mapping from single image or from multiple images.

5. CONCLUSIONS

We have found the object-oriented design as extremely helpful to redesign our system without a need for rewriting the complete existing code. By using the notation of the Unified Modeling Language we are able to describe the complexity of our interactive system in a clear way. We have redesigned the user-interface. In this way we have improved the communication between the user (operator) and the automated modules of the system. By using the Model-View-Controller design pattern we have now reached a more loose coupling of the GUI to the system. This means that it can be easily replaced, which considerably increases the portability to other platforms.

We have made good experience with the integration and adaptation of existing automated modules like the height measurement or the docking of primitives as well as with the integration of newly developed software like the automatic texture extraction. By the migration of the previous system[1] into an object-oriented system, we can observe a considerable improvement in the maintenance of the previous software.

As the system itself can be reused as an object, it is possible to apply the modeler in another context as well as to extend it by additional functionality. The extendibility of the Semiautomatic Building Extraction System allows e.g. to integrate the modules into a system, based on the Common Object Request Broker Architecture (CORBA),[11] which is proposed as a Standard by the Object Management Group (OMG). This means, the system can for certain

applications access distributed objects in the Internet or Intranet as well as being accessible as object by other objects based on CORBA.

In the case of texture extraction we plan additional functionality by combining close-range imagery with aerial-imagery for site-modeling, which means information fusion from highly different sources. In the case of matching we want to improve the existing modules to a more general matching tool, that allows to measure single points on the ground as well as linear features like street borders. In this way we can extend the building acquisition to the acquisition of other object types represented in a 3D-city model.

REFERENCES

1. R. Englert and E. Gülch, "One-eye stereo system for the acquisition of complex 3D building descriptions," *GIS* **9**(4), 1996.
2. A. Brunn, E. Gülch, F. Lang, and W. Förstner, "A multi-layer strategy for 3D building acquisition," in *Proceedings IAPR-TC7 Workshop, Graz*, 1996.
3. E. Gülch, "Extraction of 3D objects from aerial photographs," in *Proceedings COST UCE ACTION C4 Workshop 'Information systems and processes for urban civil engineering applications', Rome, Italy, November 21-22*, 1996.
4. G. Booch, *Object-oriented Analysis and Design. With Applications.*, Benjamin/Cummings Publishing Company, Inc., 1994.
5. G. Booch, I. Jacobson, and J. Rumbaugh, *Unified Modeling Language, Version 1.0.* Rational Software Corporation, v 1.0 ed., 1997. URL: http://www.rational.com.
6. W. Schickler, "Feature matching for outer orientation of single images using 3-D wireframe controlpoints," in *Internat. Archives for Photogrammetry, B3/III, Washington*, pp. 591–598, 1992.
7. T. Läbe and K.-H. Ellenbeck, "3D-wireframe models as ground control points for the automatic exterior orientation," in *Proceedings ISPRS Congress, Comm. II, Vienna, IAP Vol. XXXI*, 1996.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Addison Wesley, 1995.
9. H.-J. Bungartz, M. Griebel, and C. Zenger, *Einführung in die Computergraphik*, Vieweg, 1996.
10. A. Stepanov and M. Lee, *The Standard Template Library.* Hewlett-Packard Company, 1995.
11. Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 1996. http://www.omg.org/corba/corbiiop.htm.