

Object Oriented Analysis of Forest Growth and Yield

By
Richard Zinck

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF SCIENCE
AT THE
GEORGIA-AUGUSTA UNIVERSITY
GÖTTINGEN, LOWER SAXONY
JULY 2003

©Richard Zinck, 2003

GEORGIA–AUGUSTA UNIVERSITY
DEPARTMENT OF
MATHEMATICS AND INFORMATICS

I hereby certify that I have written this thesis myself without using any sources other than cited.

Göttingen, Lower Saxony
July 31, 2003

Richard Zinck

To Peter on his 22nd birthday.

Contents

Acknowledgements	vii
Abstract	viii
Zusammenfassung	ix
Preliminaries	xi
1 Forest growth and yield models	1
1.1 Purpose and application	1
1.2 Classification	2
1.3 Features & Problems	3
1.4 Likely Extensions	5
1.4.1 Geographic information systems	5
1.4.2 Landscape modeling	5
2 OOA of forest growth and yield	7
2.1 A word on time	9
2.1.1 A model and its time	9
2.1.2 Dependencies of model objects in time	10
2.2 Use cases	15
2.3 Static structure	18
2.3.1 The tree	18
2.3.2 The forest stand	21
2.4 Dynamic model	22

3	Model implementation and extension	31
3.1	Dealing with reference systems	32
3.1.1	Units	32
3.1.2	Geographic reference systems	33
3.2	Package structure	34
3.3	Patterns	36
3.3.1	Refining the composite model	36
3.3.2	Template Method	43
A	Terms & Abbreviations	46
	Bibliography	51

List of Figures

2.1	Time Chain I	10
2.2	Time Chain II	13
2.3	Forest management	16
2.4	Forest growth	17
2.5	Visitor Pattern	28
2.6	Visitor sequence I	29
2.7	Visitor sequence II	30
3.1	Localizable	34
3.2	Location	34
3.3	Composite Pattern I	40
3.4	Composite Sequence	41
3.5	Composite Pattern II	42

List of Examples

2.1.1 Simple 'time' model in sequential programs.	9
2.1.2 Method 'notify()' in 'ObservedSubject'.	12
2.1.3 Method 'update(o:Observable)' in ObservedObserver.	12
2.1.4 State change in a three element time chain.	13
2.4.1 Working with a forester object.	23
2.4.2 Accept method in aggregated 'Host' objects I.	24
2.4.3 Accept method in aggregated 'Host' objects II.	25
3.1.1 Solution to unit problem I	32
3.1.2 Solution to unit problem II	32
3.3.1 Forwarding of the message 'grow()' in 'GrowthModelComposite'.	38
3.3.2 Sample live() method in 'LivingObject'.	44

Acknowledgements

I would like to thank my supervisor and first reader Prof. R. Switzer for his prompt willingness to supervise a bachelors thesis with a topic which, at first glance, might have seemed far away from the area of his recent work, object oriented distributed systems.

Modelers from all disciplines have discovered the benefit of using tools and ideas from computer science. Furthermore, these disciplines provide interesting and fun opportunities for computer scientists to bring their knowledge to work. Object oriented modeling and design patterns are of great benefit not only in software engineering but also in modeling. Demonstrating this on the example of forest growth and yield models is part of my thesis.

Thank you for your constructive guidance!

Göttingen, Lower Saxony
July 31, 2003

Richard Zinck

Abstract

Forest growth and yield models are developed to assist in decision-making and model validation by providing information. The content and quality of this information varies from model to model.

Simulation models can be classified by similarities in their structure. Common types of forest growth and yield models include stand and single-tree models which can be distance-dependent or distance-independent.

Nevertheless, all types of forest growth and yield models have to deal with characteristic features of forest ecosystems and management. These include different scales in time and space, longevity, stand structure, system openness and the multitude of different interests in forest management.

There are many approaches to modeling natural phenomena. Object oriented modeling is one. The key advantage of object oriented modeling is its relatedness to the nature of human thought.

Time is often implicitly 'wired' into a program. Nevertheless, it is an important part of a model. Modeling the changing state of objects in time with a sequential program can cause problems. One such problem is the loss of information about the prior state(s) of an object while it is still needed by another object. The time-chain provides an elegant solution to this problem.

The definition of the classes which are the backbone of a model is of prime interest. The types should be useful in most kinds of forest growth and yield models to ensure compatibility while not exploding in size.

A forester, forest worker, storm or herd of animals can be seen as visitors to a stand. Interventions and other actions, even growth models, can be brought to a stand by appropriate visitors. Combining the visitor pattern with a set of common base types allows great flexibility and extensibility.

Zusammenfassung

Waldwachstumsmodelle werden entwickelt um Informationen für Waldmanagement und Modellvalidierung zu liefern. Informationsgehalt und –qualität variieren von Model zu Model.

Simulationsmodelle können über Ähnlichkeiten in ihrer Struktur klassifiziert werden. Häufig wird zwischen Bestandes – und Einzelbaummodellen unterschieden welche distanzabhängig oder distanzunabhängig sein können.

Alle Arten von Waldwachstumsmodellen müssen sich jedoch mit den charakteristischen Eigenheiten der Waldökosysteme und ihres Managements auseinandersetzen. Dazu gehören der Aufbau in verschiedene Zeit – und Raumskalen, die Langlebigkeit, die Offenheit des Systems und die Vielzahl an unterschiedlichen Interessen, welche hinter dem Management von Wäldern stehen.

Es gibt viele Wege natürliche Phänomene zu modellieren. Einer davon ist die objektorientierte Modellierung. Ihr ausschlaggebender Vorteil liegt in ihrer Verwandtschaft zur Natur des menschlichen Denkens.

Zeit wird häufig indirekt in ein Programm eingebaut. Sie ist jedoch ein wichtiger Bestandteil eines Modells. Werden Zustandsveränderungen von Objekten in der Zeit durch ein sequentielles Programm modelliert, so kann das Probleme verursachen. Ein solches Problem ist der Verlust der Information über vorherige Zustände von Objekten während sie von anderen Objekten noch benötigt wird. Die 'time-chain' bietet eine elegante Lösung für dieses Problem.

Die Definition der Hauptklassen eines objektorientierten Modells ist von großer Bedeutung. Ihre Schnittstellen sollten in den meisten Arten von Waldwachstumsmodellen nützlich sein ohne zu umfangreich zu werden.

Förster, Waldarbeiter, ein Sturm oder eine Herde Tiere können als Besucher eines Waldbestandes aufgefaßt werden. Jegliche Form von Intervention, sogar das Wachstumsverhalten, kann über einen 'Besucher' in einen Wald gebracht wer-

den. Die Verbindung von grundlegenden Schnittstellen mit dem visitor Muster ermöglicht eine große Flexibilität und Erweiterbarkeit des Modells.

Preliminaries

Aim of the thesis

The aim of the thesis is to develop a basis for an object oriented forest growth and yield model which is:

- elegant
- easy to understand, maintain and extend
- suitable for implementing many different existing models of forest growth and yield.

The work does not include the implementation of a particular model or model family. The time frame for completing this work is three months, the work load is set to 15 ECTS (10 SWS) according to faculty standards.

Thesis structure

The chapter 'Forest growth and yield models' gives a condensed overview of the purpose and application of forest growth and yield models, a classification and a list of features and problems which are characteristic for forest growth and yield models. It ends with thoughts about likely extensions of forest growth and yield models.

The chapter 'Object oriented analysis of forests growth and yield' shows use cases of typical interactions which are of interest in modeling forest growth and yield. A time model is introduced which solves problems of interdependence of model objects in time. It then proceeds to develop a static model of a forest

stand and trees. Finally, a way of modeling dynamic behavior and interactions is introduced.

The chapter 'Model implementation and extension' addresses the problem of dealing with different unit and reference systems. A solution is given based on abstracting physical properties. The chapter proceeds with describing a package structure. It ends with a discussion of further patterns which are likely to be useful when implementing a particular forest growth and yield model.

Appendix A lists terms from object oriented development and forest growth and yield modeling which are not explicitly explained in the text. A forest scientist will find information about basic object oriented terms while a computer scientist can look up the meaning of an unknown term from forest growth and yield modeling. It does, however, only provide general information. More explicit information can be found in the literature listed in the bibliography.

Addressees

There are a number of addressees who might be interested in this thesis. They include:

Forest scientists Forest scientists who work on forest growth and yield modeling might be interested in the design of the object oriented forest growth and yield model as well as the applied patterns and solutions to practical problems.

Computer scientists Might enjoy seeing object oriented techniques and patterns being applied in a unusual field.

Students Can use this work as a basis for developing their own particular forest growth and yield models.

Lexical conventions

Types and messages are enclosed in quotation marks, e.g. 'add(object c)' and 'Vector'. Furthermore, a type is written with a beginning capital letter. Examples are formulated either in UML or Java.

Chapter 1

Forest growth and yield models

1.1 Purpose and application

There are many reasons for developing a forest growth and yield model. One reason, for sure, is business. A forester needs to know about the quality, amount and location of his (natural) resources. This information can easily be stored in a database, however. It is simply a data model without behavior. Nevertheless, obtaining it is rather expensive. Furthermore, natural resources have a behavior.

A forest stand is a complex ecosystem which reacts to disturbances such as timber harvesting. How does it react? What kind of reaction is favorable, which is to be avoided? What happens in the long run if certain types of usage are employed? Another reason, therefore, is sustainable resource management.

Forest growth and yield models are developed to assist in decision-making and model validation by providing information. They are used by forest businesses and government departments in mid and long term planning. According to Lemm (1991, p. 21-22) this planning can be on:

- forest stand
- forest office or
- regional

level. The information provided needs to be suitable to answer questions of forest management. These include the following aspects:

- forest resources assessment

- timber production
- efficiency of forest utilization
- evaluation of forest damage
- comparison of management strategies

Altogether forest growth and yield models have become an indispensable tool for forest management.

1.2 Classification

Simulation models can be classified by similarities in their structure. This is achieved by isolating and comparing distinguished features of models, e.g. by asking about common problems and how they are solved. Every classification, however, is arbitrary to a certain degree, separating aspects that belong together from another point of view.

Common types of forest growth and yield models include stand and single-tree models which can be distance-dependent or distance-independent. A single-tree model derives stand attributes by aggregating them from the features of single trees. In such a model the growth of a tree might be calculated depending on the influence (distance) of neighboring trees. Another possibility is to derive a competition factor from stand characteristics. According to Lemm (1991, p. 28) forest growth and yield models can be classified by:

- ◆ element relations
 - ◆ Input / Output
 - Relationships between input and output of a system are gained by **statistical** evaluation. The modeled part of reality is seen as a 'black box'.
 - The modeled system is broken up into subsystems, whose relationships and behavior explain the behavior of the modeled system. The model is **structured**.
 - ◆ Time
 - The behavior of a **continuous** model changes only an arbitrary small amount in a small time step.

- The behavior of a **discrete** model is defined only at certain points in time. It may change its behavior abruptly from one point in time to the next.
- Models working with continuous and discrete behavior.
- ◆ Determinism
 - A simulation is **deterministic**, if the mappings of all entities are deterministic in the model.
 - If the mapping of an entity has more than one possible value and if a probability for assuming each value is provided it is a **stochastic** model.
- ◆ behavior in time
 - ◆ **static**: model elements, their topology and relations do not change during simulation. Elements may change their state, however.
 - ◆ **dynamic**: model elements, their topology and relations can change during simulation.
- ◆ degree of abstraction : The level of abstraction is defined by the elements and relations chosen to depict reality. The degree of resolution is compared to the minimal resolution which is necessary to be able to answer a given question.

1.3 Features & Problems

Forests, as natural ecosystems, have a few characteristics which should be reflected in a forest growth and yield model. Not all of these characteristics can and should be implemented in a particular model, however. Modeling utilizes abstraction from 'unnecessary' details in order to simplify what we perceive as reality, thereby making it understandable. Forest growth and yield models have a well defined purpose and should be constructed accordingly. Nevertheless, often it is of interest to adapt an existing model to answer similar questions or to change parts of the implementation. One might consider adding new factors to a diameter growth function or alter its computation altogether. It would be nice if the core

model need not be changed for this kind of adaption. Now, what might be characteristics to consider in designing a forest growth and yield model? According to Pretzsch (2001, pp. 60-85) these include:

Scale Processes in forest ecosystems can be investigated at various levels of detail in time and space. The timescale of interest may vary from seconds to thousands of years, while one may be interested in cell physiology or soil development. To make things worse, the structure and behavior of one level is influenced by the next lower one and vice versa.

→ The level which is of interest in forest growth and yield models is the forest stand. Because of their higher promise of insight, trends go toward structured models, describing stand development by modeling the changes of its trees¹, often on an ecophysiological level. A simulation model thus needs to be extensible to the next lower or higher level of detail. Nevertheless, the model goal needs to remain supported. Furthermore, the simulation model needs a mechanism for allowing things to happen on different time scales.

Longevity It takes many generations of researchers to monitor developments in forest ecosystems. The human lifespan is approximately 10^2 while that of a tree is up to 10^4 . This has to be taken into account when planning and implementing sivicultural experiments, e.g. comparing the effect of different thinning strategies.

→ The above fact illustrates why simulation models are important in forest management. Time is a crucial factor in planning and conducting sivicultural experiments. Testing all types of sivicultural interactions in all stand types is tedious and virtually impossible because of sparse resources (time, money and space) and the number of possible combinations.

Structure A forest stand is determined to a large extent by its structure. Trees are anchored to the earth by their roots, movements amount to swaying in the wind. Furthermore, their constellation influences growth factors such as radiation, temperature and precipitation.

→ A forest growth model should be able to depict stand structure.

¹Considering all trees in a stand or representatives of diameter–distribution classes.

Openness A forest stand is an open system. It exchanges matter, energy and genetic information with its environment.

→ The model needs to include a mechanism for effects coming from 'the outside', e.g. acid rain or a storm.

Interest There are many aspects to land use and forests are no exception. One might be interested in the forest's effect on nutrient cycles, production of timber, influence on local or global climate, recreational value etc. Chances are high that once a model has been constructed, new criteria of interest will evolve.

→ It must be possible to include new features in response to new needs. One might, for example, want to add a new attribute to stem or crown profiles.

That sounds like a lot. And in fact, it is. The next chapter will show how object oriented programming can be of assistance in developing a forest growth and yield model which is flexible enough to fit these needs.

1.4 Likely Extensions

1.4.1 Geographic information systems

The importance of geographic information systems in forest management has steadily increased in the recent years. This is a natural development since forest management is, after all, strongly related to land use management. Trees, and therefore forest stands, can be located by geographic coordinates, e.g. a polygon describing stand boundary. Some models, such as SILVA (Pretzsch and Kahn, 1998) allow coupling with certain geographic information systems (Pretzsch, 2001, p. 277). Thus, a future coupling of the simulation model with a geographic information system is to be expected.

1.4.2 Landscape modeling

Forest management and range management are strongly related. It is very likely that interests in a simulation model for landscape development will arise. This

might include elements such as grass lands and aquatic ecosystems next to different kinds of forest stands. Questions might also arise on how such elements, on a smaller scale, interact in making up a forest stand, e.g. how small ponds and grass patches influence forest ecosystem diversity. Hence, the model should be able to integrate different landscape elements.

Chapter 2

OOA of forest growth and yield

Subchapter 1.2 on page 2 presents a classification of forest growth and yield models as provided by Lemm (1991, p. 28). It lacks, however, the type of object oriented model. Fishwick et al. introduce the term 'Object oriented physical modeling' in their paper (Fishwick, 1996). As I understand it they classify different kinds of common model types, such as petri nets, constraint models etc. They then describe a way to implement each kind of model using object oriented concepts and provide a framework for their integration which is in itself a complex task. To achieve this compatibility they construct a generic model of 'physical' models which is used as a common reference.

Although Fishwick (1996) often emphasizes that object oriented concepts build a bridge between the way humans think and model implementation, what they implement are mappings of petri nets, differential equation models etc. to their object oriented design for such a model.

An object oriented model is a model type of its own, however. Seen from this perspective, an object oriented physical model is a model of a part of reality using object oriented concepts. The main advantage of object oriented programming is its relatedness to the nature of human thought, forcing program implementation to follow these constructs rather than vice versa. Designing an object oriented program always includes designing an object oriented model. This is especially valuable for areas where models are used for simulating purposes, such as forest management. Object oriented system development can be divided into three phases (Erler and Ricken, 2002, p. 69):

- Object oriented analysis (OOA)
- Object oriented design (OOD)
- Object oriented programming (implementation) (OOP)

The requirements of the software are determined during the phase of object oriented analysis. This is done by creating an object model. Nevertheless, the object model should not depend on platform or programming–language–specific implementation details at this stage. Platform issues, user interfaces and data management are addressed in the second stage, object oriented design. The object model is adapted to fit the needs of a chosen platform and programming language. This can, in turn, lead to changes in the object model. If, for example, the object model uses multiple inheritance, which is not supported in some object oriented programming languages such as Java, it must be modified to use interface inheritance, on which there is no technical restriction, or composition. The third stage, object oriented programming, is closely related to object oriented design which already includes implementation details. The model is implemented and tested. This, however, is not the end. On the contrary, it is the beginning of a cyclic process in which the three stages of analysis, design and implementation recur and interact until the developer(s) are satisfied. The work in this thesis is best placed in the phase of OOA.

In object oriented modeling the world is seen as a collection of interacting objects. An object is an identifiable entity which can be isolated in a certain context (Erlor and Ricken, 2002, p. 50). The context, in addition, is defined by the problem at hand, e.g. everything that is relevant in a forest growth and yield model. Reducing complexity is vital to understanding 'reality'. What we perceive as reality is mapped to a model by abstraction, reducing it to what we believe is important in the context. To determine what is important and what is not is in itself a major task. Choosing the right factors is vital especially when a model is used in forecasting future behavior. A good model is, like good software, a piece of art. The model should (Erlor and Ricken, 2002, p. 52):

- Be simple enough to be manageable as a software artifact and understandable to humans.
- Be exact enough to provide sensible information.
- Be constructed on the highest level of abstraction which is still fit to provide

answers to the given questions.

Nevertheless, even the behavior of a good model is not necessarily that of the modeled part of reality. It needs interpretation and thorough validation to be of any use.

2.1 A word on time

2.1.1 A model and its time

It is very hard to distill the essence of 'time' into a definition. What is time in computer simulation? Time is often build into a sequential program ¹ implicitly by directing control flow, e.g. in a loop. In example 2.1.1 a unit of time has

Example 2.1.1 Simple 'time' model in sequential programs.

```
for(i=begin; i<end;i++){
  // do something to the model objects
  for(u=0;u<trees.length;u++)
    trees[u].grow();
}
```

passed every time i is incremented. It is notable that 'trees[0]' has completed its operation 'grow()' before 'trees[trees.length]'. This is caused by mapping parallel developments to a sequential program. As we will see in subsection 2.1.2 on the following page this can cause subtle problems.

Speaking more generally time can be measured by a sequence of events (Gregorius, 2002, p. 8). Time T is defined in a model by specifying the events E_T which cause time to 'proceed'. Seen in this way, time is a set² of 'received' events, the moment m being the number of elements in the set. What exactly is an event? An event is a notification that something has happened, e.g. all trees in our model

¹Establishing 'simultaneity' is a major task in parallel programming.

²It is important, however, that elements can only be added to the 'set', no element added may be removed.

have received the message 'grow()'. It might also indicate that a certain state has been reached. Other events might not cause time to proceed, such as a local timber harvest or storm. These events take place 'in' time, without defining it.

If all events which can cause time to proceed originate from within the model we have an internal timer. If all events $e \in E_T$ are received from outside the model it is an external timer.

2.1.2 Dependencies of model objects in time

Modeling the changing state of objects in time with a sequential program can cause problems. One problem is the loss of information about the prior state of an object. As noted in example 2.1.1 on the page before time has changed for

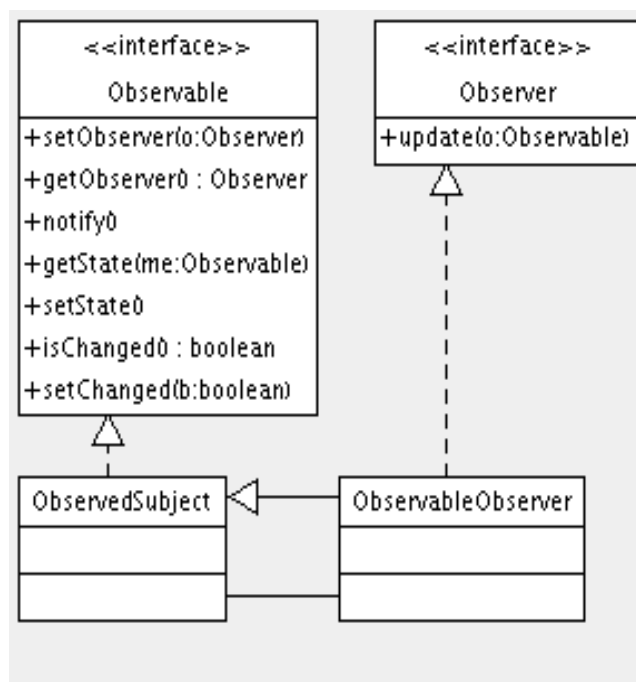


Figure 2.1: Time-chain class structure.

all objects of the simulation only after they were reached by control flow. Thus, some objects have already changed their state while others have not as yet. The information about their prior state is lost. This information might be needed by other objects which calculate their change depending on that prior state. Let N_{it}

be a set of objects whose states at time t influence the state of i at $t + 1$. The problem always occurs if

$$\exists \text{ an object } o \in N_{it} \text{ for which } i \in N_{ot}$$

If o depends on i and i does not depend on o the order in which the change is performed is crucial. If o changes before i everything is fine. However, if i changes before o does, we have a problem. A simple, yet not very elegant, solution to this problem could be to double the attributes of an object, one carrying the value of the actual and one of the prior state. There is, however, a more elegant and flexible solution, the **time-chain**. The time-chain uses a modification of the observer pattern to provide the necessary information. Figure 2.1 on the preceding page shows a class diagram. The 'ObservedSubject' implements the interface 'Observable'. It has a reference to its 'ObservableObserver' object of which it has, contrary to the standard observer pattern as presented by Gamma, Helm, Johnson and Vlissides (1995, pp. 293-304), only one. The 'ObservableObserver' implements the Observer interface. It provides a method 'update(o:Observable)' which is called by the respective 'ObservedSubject' after it changed its state. Furthermore, the 'ObservableObserver' inherits from the 'ObservedSubject'. It does so to provide two things. Firstly, it can mirror the state and behavior of an 'ObservedSubject' by using its own attributes and methods inherited from the ObservedSubject class. Secondly, it is able to be observed itself by an 'ObservableObserver'. This allows the construction of a chain of 'ObservableObservers' which can have a head object of the type 'ObservedSubject'. Figure 2.2 on page 13 shows how it works. The aim is to provide a mechanism to store information of previous object states and behavior. Every element of the chain represents the state of the head object at a certain time, making a list of subsequent states and behavior. If the 'ObservedSubject' changes its state it informs its observer via its 'notify()' method. The 'notify()' method might be implemented as in example 2.1.2 on the following page. It is, therefore, very simple. The method 'update(o:Observable)' is more interesting. The time-chain can only work if this method is implemented correctly. It is important to make sure that, if an 'ObservableObserver' has an 'Observer' itself, it forwards the message 'update(o:Observable)' to this Observer before adapting its state. This is done in example 2.1.3 on the next page. One might, of course, substitute the 'notify()' method in 'ObservedSubject' by 'update(o:Observable)'.

Example 2.1.2 Method 'notify()' in 'ObservedSubject'.

```
public void notify(){  
  
    myObserver.update(this);  
}
```

Example 2.1.3 Method 'update(o:Observable)' in ObservedObserver.

```
public void update(Observable o){  
  
    if(myObserver != null){  
        myObserver.update(this);  
    }  
    o.getState(this);  
}
```

This would make all elements in the time chain alike. I did not do so in order to have a head element. If the head element were also an observer, it could observe the tail element in its chain building a cycle. If the state of one of the cycle elements changes this could lead to perpetual cycling of 'update(o:Observable)' messages.

By first forwarding the 'update(o:Observable)' message the elements representing the oldest state in the chain, which are located at the tail, can update their state to the next following state in time, represented by the chain element before them. This information would be lost if the second and following elements changed first before forwarding the message. The method 'getState(me:Observable)' is responsible for changing the state of the object 'me', which is an 'ObservableObserver', to the same state as the executing object. Hence, the object sets the value of attributes in the 'ObservedObserver' while executing the 'getState(me:Observable)' method³.

Let's take a look at the behavior of a three element time-chain. At the begin-

³This is not shown in figure 2.2 on the following page because it differs from implementation to implementation.

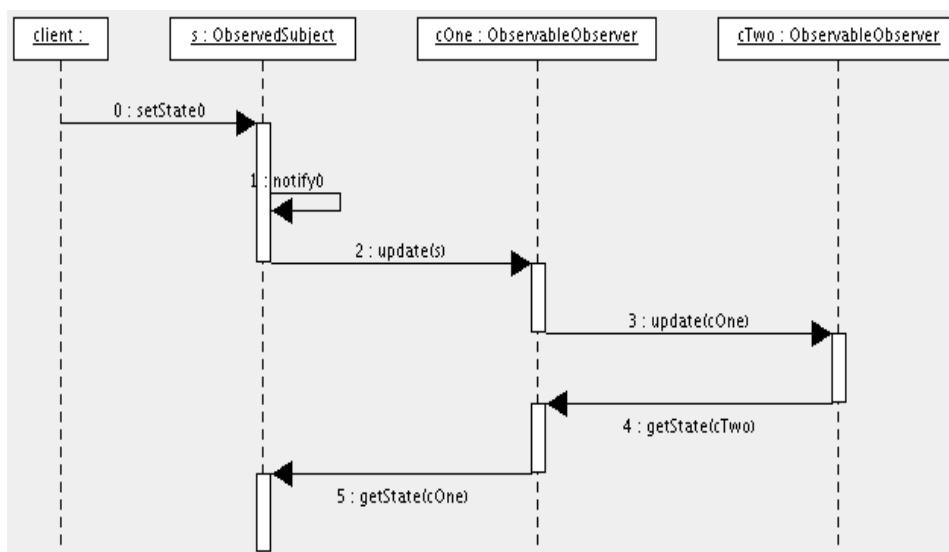


Figure 2.2: Sequence diagram for a three element time chain.

ning all elements have the same state. Changes take place in the head element, the 'ObservedSubject'. The state of a chain element is represented by a number. As we can see the time chain shows the subsequent states of the head object

Example 2.1.4 State change in a three element time chain.

```

0. Start : 0 0 0
1. Change: 1 0 0
2. Notify: 1 1 0
3. Change: 2 1 0
4. Notify: 2 2 1
5. Change: 3 2 1
6. Notify: 3 3 2
  
```

only after a change occurred and before 'update(o:Observable)' has been called. It is, therefore, crucial to define who is responsible for updates at what time during simulation. If, as in example 2.1.1 on page 9, a tree changes its state before others, and it sends the message 'update(o:Observable)' to its observers directly after the change takes place the information about the prior state would be lost

in a two element chain. In a three element chain it would be in the last element. Thus, there are two options. The first option is to let the 'ObservedSubject' send 'update(o:Observable)' messages as soon as its state has changed. This implies a three element chain. The neighboring trees have to rely on the state of chain element three of the changed tree in order to compute its influence on their development. I do not like this option because a three element time-chain raises costs in memory and time through more updates. Furthermore, a tree cannot tell if a neighbor is ahead in time. Hence, a counter or other mechanism has to be added to the tree class which provides this information. The computation of influence can then take place depending on either the head or the third element of the time chain.

Therefore, I prefer the solution of delegating the responsibility for sending 'update(o:Observable)' messages to the control flow of the model ⁴. The 'update(o:Observable)' messages have to be sent at the end of a time interval just before beginning the next. Whoever is responsible for time management in the model has to make sure that all 'ObservableSubjects' which have changed during the interval send 'update(o:Observable)' messages to their observers. This solution has the following benefits:

- There is less overhead in memory and time because a two element time chain is sufficient.
- The object responsible for making the 'ObservedSubject's send 'update(o:Observable)' messages can select what kind of changes need to cause an update. This can improve efficiency by reducing unnecessary calls.
- The second element has the same state as the 'ObservedSubject' at the beginning of each time interval. It still has this state after the ObservedSubject chaged until notified, thus during the whole time step. Therefore, neighboring trees or other depending objects can always use the second time-chain element to perform their calculations without needing to know whether the 'ObservableSubject' has changed or not.

A possible disadvantage is that a client might forget to send the 'update(o:Observable)' messages. Hence, all update relevant information should be clearly documented.

⁴In extreme, this might be a change manager.

Memory allocation and deallocation should be easy for a time chain, solved in a similar way to the forwarding of 'update(o:Observable)' messages. The class structure 2.1 on page 10 relies on the pull model because after an initial change in the 'ObservedSubject', the state changes take place 'backwards', from the last chain element to the second. These elements do not remember how exactly they changed the last time, therefore making a detailed push model unpractical.

2.2 Use cases

Object oriented analysis starts with a rough system analysis. What are the actions of interest, who are the actors and how do they come together? Figure 2.3 on the next page shows a use case diagram for sivicultural action and actors. It is purposely kept simple. Nevertheless, it shows the most common actions and actors relevant to a forest growth and yield model.

Why start with a use case diagram in forest utilization? Forest growth and yield models are constructed for a well defined purpose (see subsection 1.1 on page 1). This purpose determines which aspects are of importance in modeling the growth of a forest stand.

The abstraction takes place from the point of view of a forester. An ecologist has a different view and so has a tree physiologist. Nevertheless, the view of a forester includes aspects of ecology, management, politics and timber utilization. A forester is interested in biomass production as well as in profit per solid meter. This diverse range of interest must be adequately reflected in the design of the model.

It is common practice to add new features to existing forest growth and yield models as they come. There is nothing wrong in principle with that. The crucial point is that the cost of such modifications and additions in late phases of the software life-cycle are a lot higher than at the beginning. Hence, a sound design and analysis at an early stage can reduce future costs enormously (Gamma et al., 1995). Furthermore, the design becomes more elegant, clear and structured which helps in the maintenance process.

As figure 2.3 on the next page shows there are relationships among actions. Thinning, as an example, can be seen as a specialized tending intervention, therefore establishing an is-a-relationship. Of course there are further subtypes to thin-

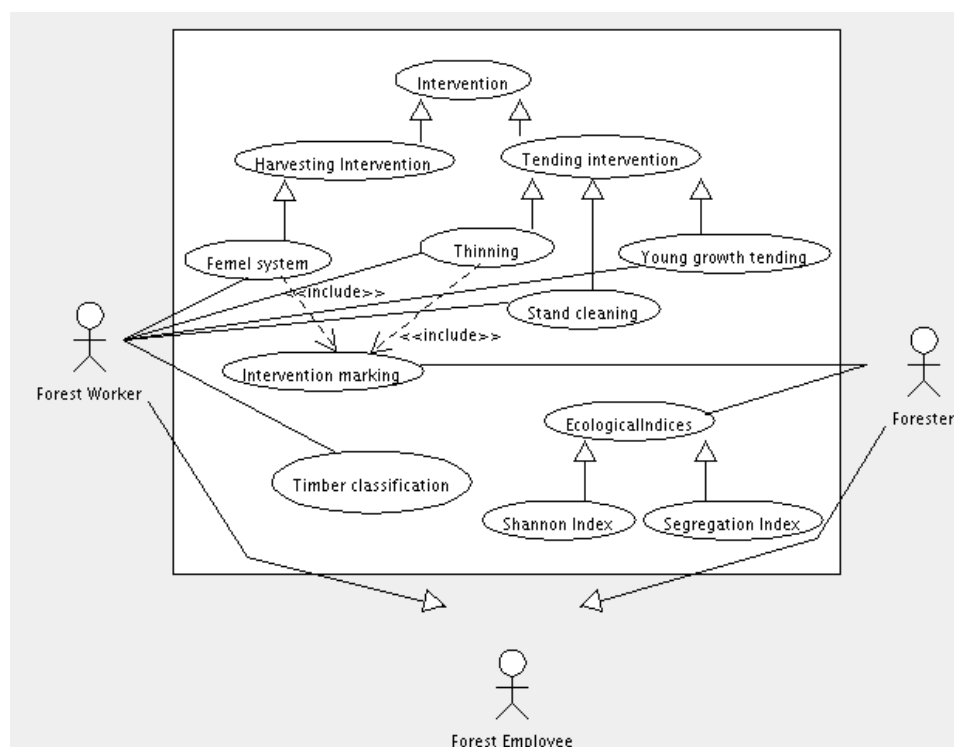


Figure 2.3: Simplified use case of forest management.

ning, such as high and low thinning, which are not included here. Cleaning stands, tending young growth, afforestation etc. all can be seen as specialized tending interventions.

In Germany these interventions are done by forest workers under the supervision of a forester. Both forest workers and foresters are forest employees. The actor, in this case a forest worker, is connected to the corresponding actions by a line. Some actions, such as harvesting and thinning, require intervention marking by a forester who is responsible for the selection of crop trees. The actions femel system and thinning use intervention marking, depicted by a serrated line. Foresters pride themselves on sustainable resource management. It is, however, not a simple task to determine the 'degree' of sustainability of a certain type of forest utilization and management. Furthermore, interest in diversity measures has increased. Every index, however, covers only parts of what is associated with the term 'diversity'. There is diversity in species, structure, genetic information etc.

Hence, the design of a forest growth and yield model should include an elegant mechanism for the support of such indices.

Figure 2.3 on the preceding page provides a basic overview of actions whose effects are of interest in a forest growth and yield model in the eye of a European forester. Although not complete it should suffice as a basis for selecting important aspects while abstracting forest growth under the influence of interventions. The level of abstraction employed in analysis is relatively high because I am not designing a particular forest growth and yield model. The forest stand, as an aggregated object of trees (and maybe soil, etc.) is typically not seen as a living organism but as an ecosystem. Life Processes, therefore, pertain to the trees. Basically, they include maintenance, growth, reproduction and death. How any of

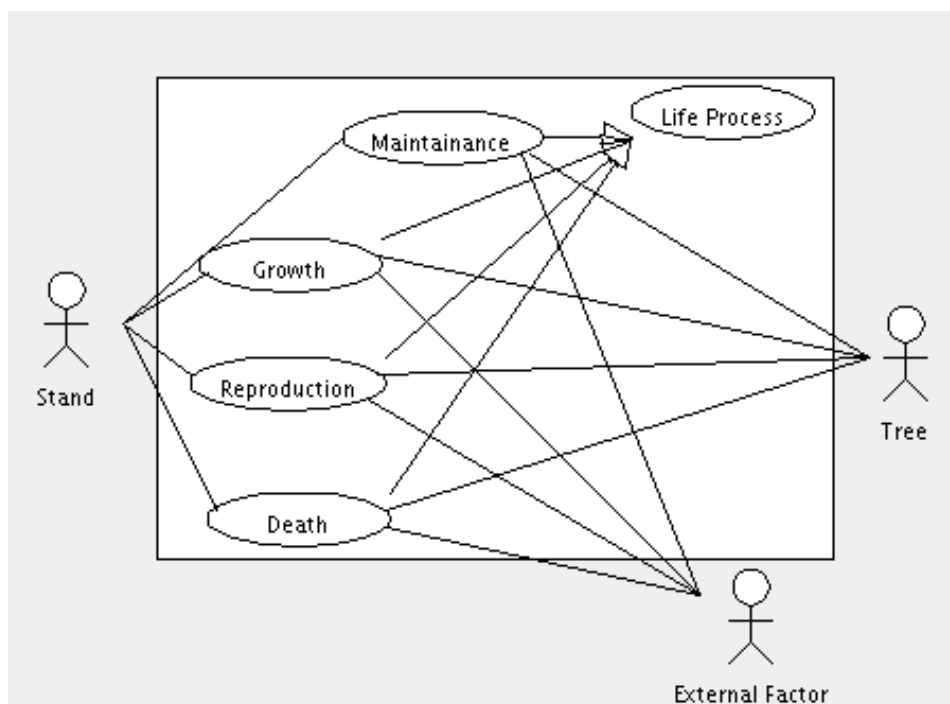


Figure 2.4: Simple use case of forest growth.

these actions are modeled by a particular forest growth model is not of interest here as long as it can be technically supported. The interesting aspect is that the life processes of a tree are influenced by other trees, stand factors such as local soil characteristics and other factors depicted as 'External Factor'. Every forest

growth and yield model is likely to include factors which another does not or see their influence differently. Thus, the design should allow for maximum flexibility and extensibility with regard to modeling the life processes and the effect of external factors.

2.3 Static structure

The definition of the classes which will be the backbone of a model is of prime interest. Obviously a forest growth and yield model will need a forest stand and trees. The interesting question is how high the degree of abstraction needs to be. In section 3 on page 31 we will see how reusability and extensibility can be provided by using appropriate design patterns and inheritance. These concepts allow the model designer to adapt provided classes to fit the needs of a special model. Thus, we can focus on basic types which find application in most forest growth and yield models here. These are, beyond doubt, the types 'Stand' and 'Tree'. Furthermore, there has to be a way to describe the actions of use case 2.3 on page 16. These include interventions, harvesting, forest value assessment, timber classification and ecological indices.

Types, rather than implementations, are described because this investigation does not target a concrete forest growth and yield model. These types can be extended to fit the needs of a special model. It is important to allow for maximum flexibility. Thus, the number of types is reduced to the necessary minimum. However, this flexibility should not come at the cost of making reusability and model compatibility impossible.

Let's begin with the type tree.

2.3.1 The tree

Trees grow. They provide timber of different quality, are influenced by and influence the forest ecosystem. Trees have a crown, branches, leaves, a stem and roots. Furthermore, they interact with the surrounding atmosphere and earth. There are, depending on the point of view taken, numerous ways to model a tree.

- Every forester is, however, going to be interested in the tree shaft. This includes information about volume with and without bark, diameter at dif-

ferent heights, for sure at breast height, and the height of the tree.

- He will further be interested in the species and age.
- The location of a tree must be known in distance–dependent models such as SILVA (Pretzsch and Kahn, 1998).
- Some models include crown dimensions and form. Others might include information about roots.

From the practical point of view the most pressing questions in designing the type 'tree' are:

1. Must stem, crown and eventually roots be provided as mandatory objects?
2. Does biological systematics need to be depicted by inheritance hierarchies? If so, how detailed?

Answering these questions is part of the evolutionary process of object oriented model & software development⁵. As for tackling the first question I suggest to take a middle way. The information which is of prime interest to forest growth and yield models as listed above should be answered by every tree. A class C implementing such an interface can forward messages to interior objects if appropriate (see figure 3.4 on page 41 for an example) and add new ones. Furthermore, let B_1 stand for the set of methods of the type 'Tree' in model 1. There are two criteria which should be taken into consideration while making the selection. On the first hand, the type should be useful in most forest growth and yield models, hence it must provide attributes and methods which are common to many models.

$$B_{common} = B_1 \cap B_2 \cap \dots \cap B_n$$

Where n is the number of models. On the second hand, it should have enough attributes to be useful in many models while not exploding in size. As a starting point I suggest the following attributes:

1. Attributes
 - (a) Age of the tree.
 - (b) Total tree height.

⁵Hence, the following gems of wisdom are most probably subject to change.

- (c) Diameter at breast height (BHD).
- (d) Diameter at seven meters height (d7m).
- (e) List of objects which influence the behavior of the tree, e.g. neighboring trees.
- (f) A reference to its stand.
- (g) A reference to an object of type 'Quality' which gives information on quality related issues.
- (h) A list of geographical coordinates defining its boundary.

2. Methods

- (a) Methods for accessing and setting attribute values.
- (b) A method 'live(int numberOfVegetationPeriods)' which implements the models view on tree growth, development and death for a number of vegetation periods.
- (c) A method 'Unit calcDiameterAt(Unit height)' which gives the diameter at a certain height. There should be two methods of this kind, one including bark and one not.
- (d) A method 'Unit calcVolume(Unit lower, Unit upper)' which calculates the trunk volume between the marks upper and lower. Again, two methods may be supplied for calculating volume with bark and without.

Clients are restricted to calling methods provided by the type 'Tree'. Nevertheless, a client who knows of the interface T_C of a specialized implementation can send all publicly accepted messages and thereby take advantage of added attributes and behavior. The second question is a bit more tricky. The species of trees which are of importance in a model can be realized by directly implementing the 'Tree' interface. The question at hand is how big the benefit of an inheritance structure is. This depends heavily on the model. Thus, it should be decided by the model developer. The problem lies in finding subtypes to 'Tree' which make sense for most forest growth and yield models. The most likely candidate is differentiating angiosperms from gymnosperms. Nevertheless, I do not see a reason for making

this differentiation at the moment. The type hierarchy can, however, be extended should the need arise. It is probably more interesting to think of extensibility with regard to adding different species of plants, such as grasses and ferns which might be included as part of the ecosystem. The type 'GrowthModelObject' in example 3.5 on page 42 is a possible solution. A framework or client only needs to know this interface to work with all 'living' objects of the model. Nevertheless, there is a number of tree genera which are common in European forestry. These include the genera *Picea*, *Abies*, *Pseudotsuga*, *Larix*, *Robinia*, *Tilia*, *Betula*, *Fagus*, *Quercus* and *Fraxinus*. It is not common to model all members of a genus, e.g. *Picea abies*, *Picea pungens*, ..., etc. It is, however, useful to provide a type for each genus. These types can be defined as abstract classes which implement attribute accessing methods etc. but leave room for implementing individual behavior.

2.3.2 The forest stand

One of the earliest approaches to modeling forest growth and yield was by yield-tables. In Germany the first yield-tables were constructed at the end of the 18th century by G.L. Hartig and J.C. Paulsen (Lemm, 1991, p.26). In the meantime, many models have been designed using aggregated stand values, e.g. BWINPro (Nagel, Albert and Schmitt, 2002) which is developed at the forest research station in Lower Saxony. Many other models use aggregated stand values such as top height, lorey height, mean diameter etc. These approaches use stand values to describe the growth of trees by regression analysis. An example of an often used attribute is the site-quality-class which is derived from the average age and height of a stand⁶.

It is not necessary to provide a classification of stands because they are aggregated objects. Most of their characteristics are defined by the state of their components attributes. Hence, things change. An even aged, pure forest stand can turn into a mixed stand. This can happen naturally or by appropriate interventions. Nevertheless, many models make assumptions about the forest stand. It is common that models, such as FBSM (Lemm, 1991), are only valid for pure, even aged stands. The responsibility for ensuring correctness must be implemented

⁶Though not for a mixed stand. This measure is quite obsolete in my eyes but still widely in use for practical reasons.

with the model by the means of the chosen language. Another problem is scale and resolution of the model. One might want to model a forest stand composed of soil, atmosphere, trees and so on. The type must be flexible enough to allow for this kind of modeling. Furthermore, it might be of interest to define a type 'LandscapeElement' or 'Ecosystem' to which 'Stand' is a subtype in order to provide extensibility for landscape simulation models. The following list contains attributes and methods for a basic type 'Stand'.

1. Attributes

- (a) Age of the stand.
- (b) A list of trees belonging to the stand.
- (c) Surface area.
- (d) Basal area.
- (e) Top height.
- (f) A list of geographical coordinates defining its boundary.
- (g) (A reference to its landscape.)

2. Methods

- (a) Methods for accessing and setting attribute values.
- (b) A method 'model(Unit timeSteps)' which is implemented from an interface 'LandscapeElement'. This method defines the stand behavior according to a particular model.

2.4 Dynamic model

Let's take a look at use case 2.3 on page 16. According to this simplified use case the forester is responsible for intervention marking and ascertaining sustainability, e.g. by inventory and ecological indices. Thus, one might get the idea of implementing a 'Forester' object which is responsible for these actions. Working with a 'Forester' object could look like example 2.4.1 on the following page. The example also shows a weakness in this approach: the forester object must implement different strategies for marking a stand. Furthermore, there are different

Example 2.4.1 Working with a forester object.

```
Forester forester = new forester();  
forester.mark(Stand s);
```

approaches to describing and defining tending and harvesting interventions. The selection of trees might be rule based such as:

'If a tree has a thick diameter and a low competition pressure then the need of a tending intervention for this tree is low (Pretzsch, 2001, p. 232).'

Other possibilities include selection according to a given stemcount-diameter-distribution which is aimed at by selecting trees to fit the distribution. Furthermore, the use case shows that there are many types of interventions with relationships such as 'is-a' and 'includes'. Implemented in a forester object, this would lead to creating several methods such as:

- markForLowerThinning(Stand s)
- markForFemelHarvesting(Stand s)
- ...

The number of methods is virtually unlimited. The situation for a 'ForestWorker' object is analogous. To make things worse, the model of which object is responsible for what actions is likely to change. This might be a consequence of structural changes in management and organization of forest offices and businesses. Hence, it is not a good solution to focus on the actors for implementing the actions.

Foresters, forest workers etc. can be seen in a more abstract way as 'visitors' to a stand. The appropriate actions are taken during the visit of a stand. This approach has many great advantages:

- The model of a visitor going into a stand is very intuitive. It fits to most models of forest management and sivicultural interventions.
- The question of which actor is responsible for what action is decoupled from the action itself. Thus, they can be reused more easily. A forester might assume the role of a 'StandThinningVisitor' or 'StandValueAssesingVisitor'. A part of his behavior can be defined by a set of roles he can assume. This allows for changes at runtime. Furthermore, an object such as a forester is not necessarily needed at all.

- It is possible to design model appropriate hierarchies of interventions. This increases order and understandability as well as code reuse through inheritance, e.g. a visitor 'FemelHarvestingVisitor' might inherit from 'HarvestingVisitor'.

Figure 2.5 on page 28 shows a sample class diagram. The 'Visitor' type declares a visit method for each 'Host' type it is supposed to pay a visit. The method signature helps to identify the 'Host' which actually send the visit message. Therefore, a 'Visitor' object has all the information it needs (type & reference) to work with the 'Host' object. However, many different 'Visitor' objects can visit a stand. Each 'Visitor', for example a 'HarvestingVisitor', implements the 'Visitor' interface, providing it with the appropriate behavior. The objects of type 'Host' declare a method accept, which gets a reference to a 'Visitor' object, e.g. a 'FirstModelVisitor'. In its accept methods each implementation of the 'Host' type needs to call the appropriate visit method of the 'Visitor' object pertaining to its type, which then interacts with the host. In figure 2.5 on page 28 the host may be an aggregated structure. This differs from the visitor pattern as presented by Gamma et al. (1995, pp. 331-344) where only the elements of an object structure declare an accept method. Not doing so allows the 'Visitor' to work with aggregated objects whose elements are again aggregated objects and so on. The accept method might look like example 2.4.2. Figure 2.6 on page 29 shows the respective sequence diagram. The 'Visitor' in class diagram 2.5 on page 28 therefore has visit methods

Example 2.4.2 Accept method in aggregated 'Host' objects I.

```
public void accept(Visitor v){
    myFirstComponent.accept(v);
    mySecondComponent.accept(v);
    // the whole is more than the sum of its parts!
    v.visitMe(this);
}
```

for all aggregates and their components. Sometimes this arouses a problem: the number of methods implemented in a 'Visitor' is equal or greater than the number of 'Host' objects it visits. If there are many 'Host' objects whose components

consist of 'Host' aggregates the visitor class can get too big. Thankfully there is a simple solution. Let every 'Host' object which is an aggregated object have its own 'Visitor' class. The chain begins with an aggregated 'Host' object. This object receives a 'Visitor' object via its accept method. But, contrary to example 2.4.2 on the preceding page, it does not forward this 'Visitor' object to those 'Host' objects which are themselves aggregated of 'Host' objects. An appropriate 'Visitor' object is created and handed over to the aggregate. Example 2.4.3 and figure 2.7 on page 30 illustrate this for the accept method. If a 'Host' object is not

Example 2.4.3 Accept method in aggregated 'Host' objects II.

```
public void accept(Visitor v){
    // Non aggregated hosts
    myFirstComponent.accept(v);
    mySecondComponent.accept(v);
    // Aggregated host
    ThirdAggregateVisitor th = new ThirdAggregateVisitor();
    myThirdComponent.accept(th);
    // the whole is more than the sum of its parts!
    v.visitMe(this);
}
```

aggregated, it simply calls the appropriate visit method of the 'Visitor' object and gives it its reference. If it is aggregated an appropriate 'Visitor' object is created and handed on. This can reduce the size of a 'Visitor' object reasonably. Furthermore, the programmer knows as a rule that every aggregated object has its own 'Visitor' class. The 'Visitor' object traverses a tree consisting of different objects, interacting with each object as needed (Gamma et al., 1995, p.336) . This has many advantages.

- The object structure which is defined by a 'Stand' class is static between vegetation periods. Furthermore, attribute values change only slightly after a vegetation period. The behavior of this structure is the core of every forest growth and yield model. Why not strip the structure of part of its behavior? Hence, a growth model could be capsuled by appropriate 'Visitor' classes.

For simple models, one 'Visitor' is likely to suffice.

- Different kinds of 'Visitor' objects (models) can interact with the same 'Host' object, e.g. the forest stand. One visitor might describe growth as in the model SILVA (Pretzsch and Kahn, 1998), the next visitor might implement the growth model of FBSM (Lemm, 1991). If the models work with the basic 'Stand' and 'Tree' interfaces this can happen at runtime. Furthermore, a new model can be developed rapidly by designing new 'Visitor' classes and using the infrastructure (framework) of another model.
- If somebody wants to use the 'Stand' structure for a different purpose than modeling growth, all that needs to be done is to write a new 'Visitor'. All kinds of interactions, such as harvesting, thinning, assessing stand value etc. can be implemented as a 'Visitor'. These 'Visitor' classes can inherit from each other, if needed. Nevertheless, interaction is not only limited to human actions such as sorting timber, all kinds of external effects, such as acid rain or a storm can be modeled by a 'Visitor' object coming to the forest stand!
- Visualizing forest stands and landscapes is important in planning land use scenarios. Different kinds of 'VisualizationVisitor' objects can run through a stand, creating 2D or 3D images.
- When extending a forest growth and yield model to a landscape simulation model, all that needs to be done is to define a 'Landscape' as an aggregated 'Host'. Now, a 'Visitor' class can be written which encapsulates the landscape model & 'Visitors'. This should allow the reuse of large parts of the forest growth and yield model (framework).
- Maintaining a model is easier because the code is concentrated in a 'Visitor' class. Even if the 'Host' objects behave as in example 2.4.3 on the page before maintaining and changing the model is easy because of the rule that every aggregated 'Host' object has a 'Visitor' object. One should not, however, forget to use a clear package system which keeps logically related classes together. One possibility is that every model implementation by 'Visitor' classes is located in its own package (see subsection 3.2 on page 34).
- As discussed in section 2.1 on page 9 time can be defined by a set of event types. The visit of a forest stand can be seen as such an event. Not all 'Vis-

itor' objects, however, need to cause model time to increase. By selecting certain 'Visitor' types, e.g. a 'GrowthVisitor', the model gets a clear time structure.

With all advantages of applying the visitor pattern to forest growth and yield models one should be aware of the following:

- If a new 'Host' class is added, whether an aggregated one or not, a new visit method has to be implemented in all affected 'Visitor' classes. Thus, the more stable the 'Host' classes are the better.

Nevertheless, the types 'Stand' and 'Tree' are very likely to provide this stability. All in all, the visitor pattern is an invaluable design feature for forest growth and yield models.

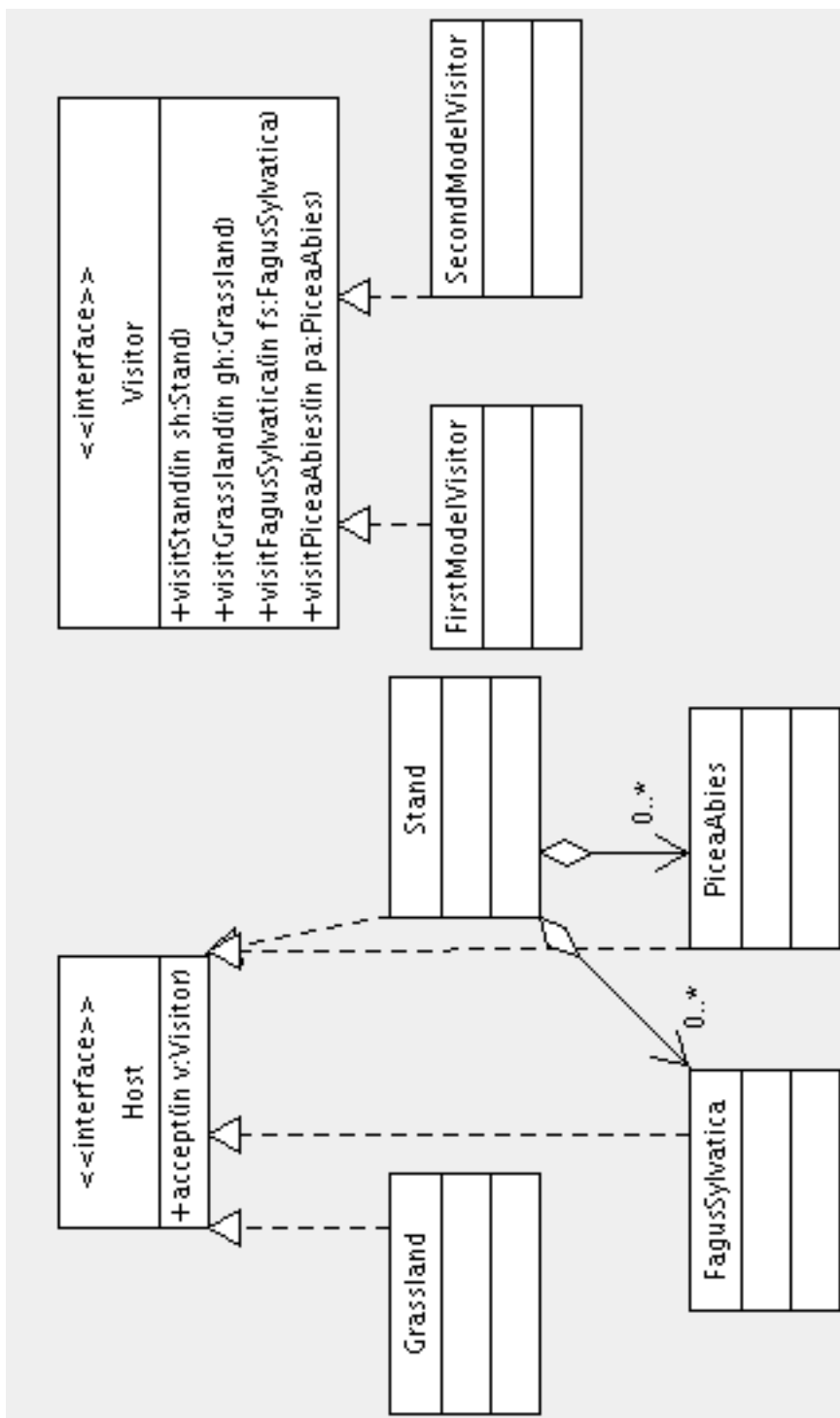


Figure 2.5: Visitor pattern modified for forest growth and yield models.

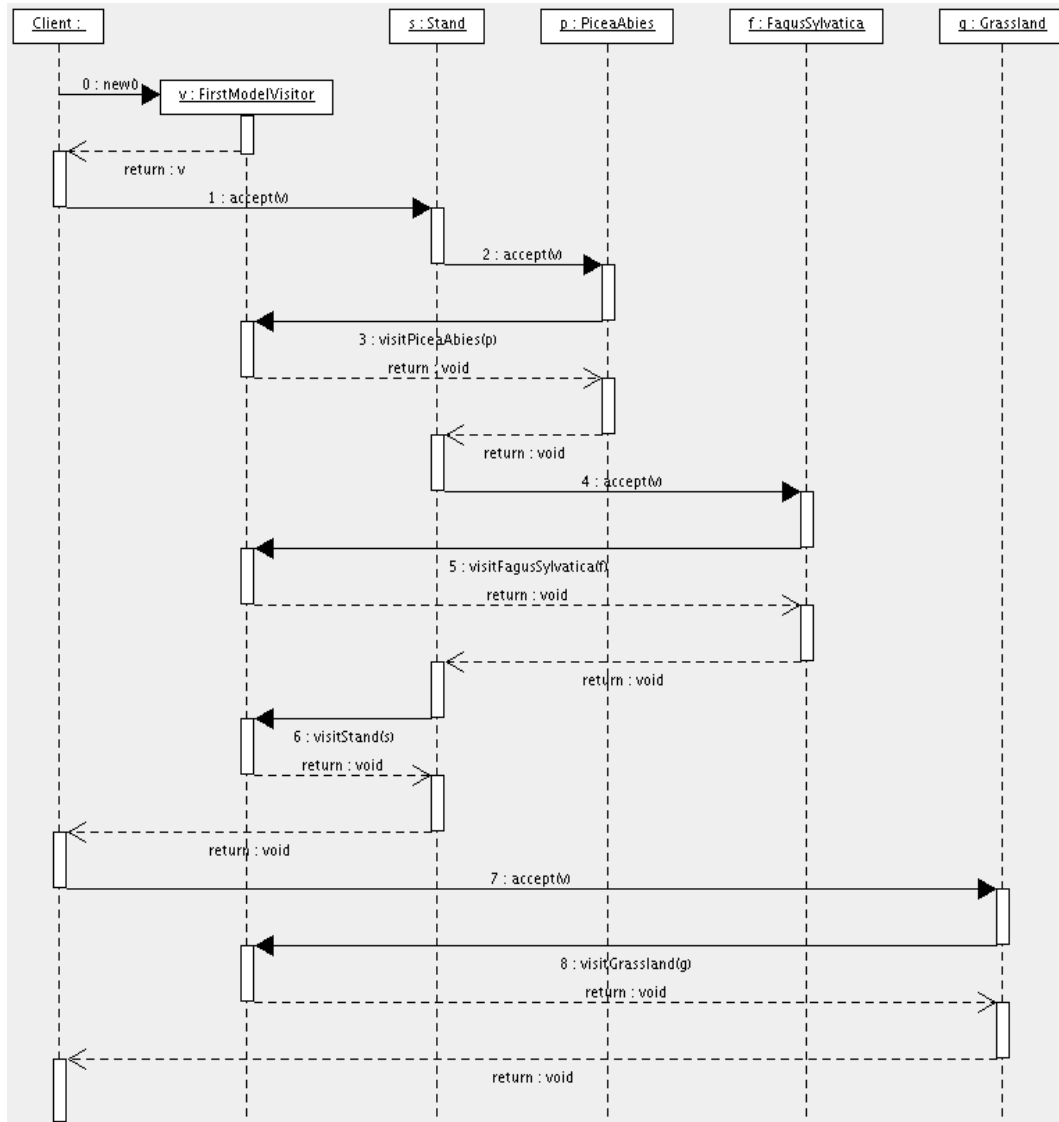


Figure 2.6: Visitor sequence according to example 2.4.2 on page 24.

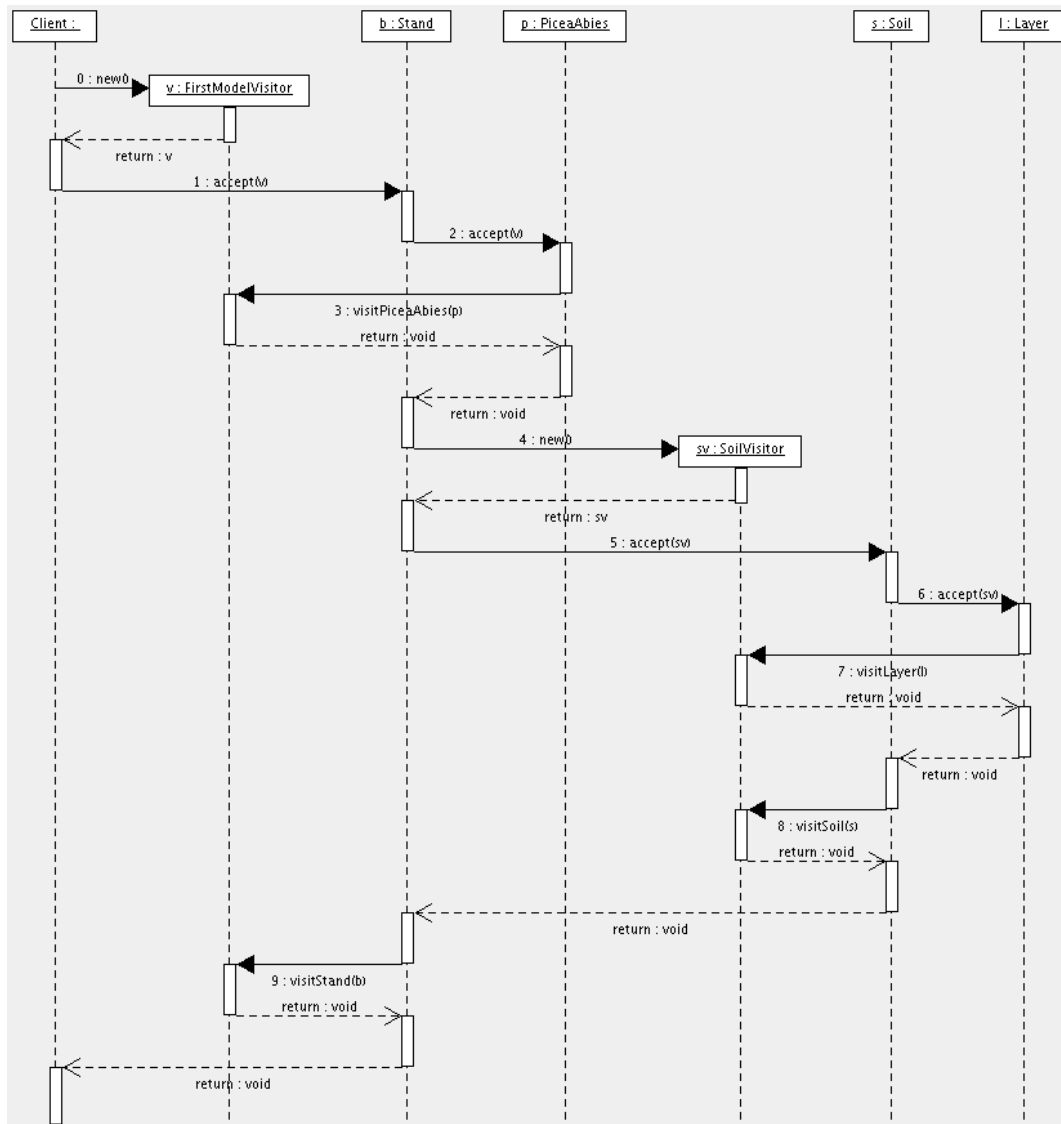


Figure 2.7: Visitor sequence according to example 2.4.3 on page 25.

Chapter 3

Model implementation and extension

Let's say you want to implement a forest growth and yield model using the visitor pattern as described in subsection 2.3 on page 18. The common types 'Stand' and the abstract classes extending 'Tree' have to be implemented or extended to fit the needs of your model. The next step is to clarify the dependencies of the model objects in time in order to find out which objects need to be head of a time-chain.

It is necessary that you have a detailed description of the behavior of your model. This behavior is then encapsulated in appropriate visitor classes. The model designer has to decide how much of an objects¹ behavior he wants to delegate to a visitor. The next step is to encapsulate the interactions, such as timber harvesting, in visitor classes. Now you can head on to write your executable program.

The combination of using visitors and interfaces for common model objects allows for a large amount of reuse. Visitors written for one model which are based on the common interfaces can readily be utilized in another such model. Compatibility, however, has to be checked by the model designer in syntax and semantics. The integration of a visitor needs to be technically possible and make sense in the respective model. It is likely that semantic compatibility will often not be the case.

¹Most likely the tree classes.

3.1 Dealing with reference systems

3.1.1 Units

Forest growth and yield models are developed to provide information for the management of forests. This information is quantified by a number of units defined by a unit system. The state of a stand or tree object is defined by the momentary value of their attributes. Furthermore, these values are modified by methods. The problem which arises here is that there are a multitude of different unit systems for quantifying a physical characteristic, e.g. a distance can be given in yards, meter or chains (ever heard of it?). To make things worse, a method yielding the diameter at breastheight of a tree might give this value in cm, dm or meter. This looks like a fine basis for hard to track semantic errors, especially if you have not programmed the classes yourself. Example 3.1.1 shows a solution building on programmer's discipline. The programmer specifies the unit of the outgoing value

Example 3.1.1 Solution to unit problem by programmer's discipline.

```
public double calcDiameterInCM(double heightInM) {  
    ...  
}
```

in the name of the function and the unit of the incoming values in the names of the signature variables. This is, however, a rather unelegant solution. A different and more elegant solution is to include a unit system in the design. This can be done by providing classes for physical characteristics such as length, surface, volume, temperature etc. Example 3.1.2 gives an alternative implementation of the above method. In this approach a unit, such as a meter, is seen as an attribute

Example 3.1.2 Solution to unit problem by inclusion in design.

```
public Length calcDiameter(Length height) {  
    double theHeight=height.getLengthInMeter();  
    ...  
}
```

of the physical characteristic, in this case 'Length'. The same can be done for surface, volume and temperature. It is, of course, tedious to write such classes for physical characteristics, but well worthwhile. A class 'Length' can be designed to provide its value in different unit systems, e.g. example 3.1.2 on the page before might as well have called 'theHeight = height.getLengthInYard()'. The costs of using classes which represent physical characteristics depends on implementation details. Nevertheless, the indirection will slow things down a bit at runtime. The 'Length' class might use a central variable to store its value in a fixed unit system and calculate the value for different different unit systems. This implementation reduces the efficiency when using a non central unit system or when working on different scales in a unit system.

Another problem is detail in range, e.g. if the value is stored internally in meter and the storage space allows for three digits greater than zero the value of one kilometer cannot be represented². Nevertheless, if a programmer experiences a problem with a given class he can extend it to improve performance depending on the problem at hand. Using classes to represent physical characteristics eliminates the problem of having to know which unit system is used by a method or attribute at which scale. More importantly it improves transparency when dealing with units.

3.1.2 Geographic reference systems

Subchapter 1.4.1 on page 5 states that forest growth and yield models are likely to be coupled with geographic information systems. This requires the simulation objects to be localizable. Figure 3.1 on the next page shows an interface which allows to locate an object by its boundary. The container class 'GeoPolygon' holds objects of type 'Location' which provide the actual information about the geographic location of a point. Figure 3.2 on the following page shows such a class for the Gauß-Krueger reference system which is in use in Germany at the moment. The type 'Location' can be extended to fit the need of the UTM (Universal Transversal Mercator) reference system, which is used by the NATO. It can further be enhanced by operations such as 'calcDistance(Location other)' etc. to make the use more practical.

²In a number system based on 10.

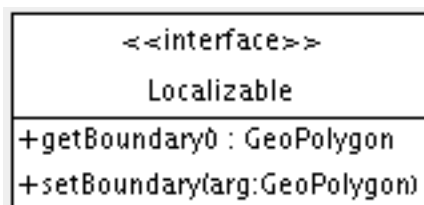


Figure 3.1: Interface for locating an object by its boundary.

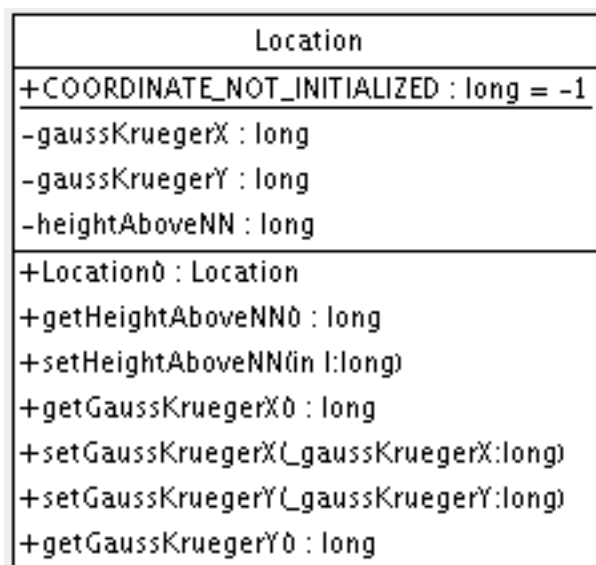


Figure 3.2: A class for locating an object in the Gauß-Krueger reference system.

3.2 Package structure

Breaking up a system into parts reduces complexity. The parts can be organized as packages which communicate with each other via interfaces. Packages depend on each other if their elements do. Therefore, too many too small packages introduce their own complexity. If, in contrary, a package is too big it will be very complex in itself. Thus, finding the right number and size of packages is very important. As a starting point I suggest the following package structure:

core The `core`³ package includes interfaces and abstract classes which are central to the model. This includes the types 'Tree', the abstract tree genera classes, the type 'Stand', an interface for a 'TimeChainHead' and an interface for a 'TimeChainElement'.

core.visitor This package contains an interface for a standard 'GrowthModelVisitor' and a 'GrowthModelHost'. It can contain other visitor and host interfaces or abstract classes which provide additional functionality for interactions⁴.

core.geo The `core.geo` package provides objects for describing the geographic location of model objects. To begin with these include the classes 'Location', which describes a point in a geographic reference system⁵, a 'GeoPolygon' which is an ordered collection of locations and an interface 'Localizable', which makes sure that every implementing object has methods to set and access a 'GeoPolygon' describing its boundary.

core.unit The `core.unit` package defines classes for physical characteristics of model objects which are quantified in a unit system. The abstraction allows for greater flexibility and more important for better transparency when using different unit systems or working with multiple scales within a unit system.

A framework or set of different frameworks can be provided in a `core.framework` package if desired. The model implementer can use an analogous, separate package structure to organize a particular model. This is helpful if you plan to implement different forest growth and yield models based on the same set of types⁶. Furthermore, the management of different versions of a model becomes easier if it is organized in a (sub)package system which keeps the classes of a particular model together⁷.

³It is common to name a package uniquely. The name 'core' can be seen as a variable for a unique name.

⁴The need for such classes is likely to arise during the cyclic development process.

⁵Subtypes for different geographic reference systems can be implemented if necessary.

⁶Depending on the goals of the modeler the 'Bridge' pattern might be worth a closer look.

⁷Once different models are implemented the 'Abstract Factory' pattern might come in handy.

3.3 Patterns

Code and design can be reused through toolkits and frameworks. Designing a toolkit or framework to provide optimal reusability is a complex task. This is especially true for frameworks, where the designer claims one design to be suitable for all programs of a certain category (Gamma et al., 1995, p. 27). Patterns are design features which are useful in solving a number of abstract problems encountered in software development. According to Gamma et al. (1995, pp. 27-28) patterns differ from frameworks in the following ways:

1. Patterns are more abstract than frameworks. Frameworks are actually implemented in a programming language, they are executable programs.
2. A pattern is a smaller building block than a framework. Frameworks often use patterns while a pattern cannot use a framework.
3. Frameworks are more specialized, they are intended for use in a special area of application. The use of patterns is not restricted to an area of application.

Nevertheless, the use of design patterns in the construction of a framework helps to make it more flexible and reusable.

The benefit of applying the observer and visitor pattern has already been demonstrated. Hence, it makes sense to investigate which other design patterns are suitable for use in a framework (or toolkit) for forest growth and yield models.

3.3.1 Refining the composite model

Problems addressed in forest growth and yield models:

- Scale
- Flexibility & Extensibility

As discussed in subsection 1.3 on page 3 forest ecosystems are hierarchically structured. A structured model covers more detail in simulating stand behavior. An ecophysiological model might, for example, model the increase of tree biomass depending on factors such as root, stem and crown respiration and leaf photosynthesis. In this model a tree exists of the elements root, stem, crown, leaf and

branch in the simplest case. This depiction is somewhat more complex than the traditional view of loggers interested in tree height and diameter. How can a framework integrate these different kinds of approaches? One possible answer is to use, or adopt adequately, the pattern composite⁸. Let's say the framework calls a routine 'grow()' on every tree object in a stand. Taking the logger's model, growing means changes in tree height and diameter under the influence of aggregated stand competition factors or neighboring trees. The physiologist's model is more complex. The growth of a tree includes changes in leaves, stem, branches, roots etc. with competition on various levels. One might, with reason, argue that this is a single tree growth model which deserves a whole new design. However, the logger's model might need extension in this direction at some time in order to meet new requirements.

The component pattern offers the following solution modified from Gamma et al. (1995, pp. 163-173) : the idea is to define an object structure in form of a tree using composition. Define an interface, or, incase you want to add default behavior, an abstract class 'GrowthModelComponent'. Furthermore, define a class 'GrowthModelComposite' as a subclass of 'GrowthModelComponent'. A 'GrowthModelComposite' has sons of the type 'GrowthModelComponent'. Because 'GrowthModelComposite' is a subtype of 'GrowthModelComponent', it may be such a son. This is how the object tree is build. The recursion is terminated by elements which directly implement 'GrowthModelComponent' and have no sons. Turn to figure 3.3 on page 40 to see a class structure ⁹.

'GrowthModelComponent' defines an interface for every object in the composition. The actions of use case 2.4 on page 17 are chosen as sample methods. You will further find methods for adding, accessing and removing sons which provide a mechanism for composition. It is important to notice that class 'GrowthModelComposite' implements the methods of 'GrowthModelComponent'. It provides a mechanism for adding, removing and accessing sons and, more important, forwards calls to the other methods to these sons. Example 3.3.1 on the following page demonstrates this for the method 'grow()'. The classes 'Stem', 'Leaf' and 'Branch' directly implement the methods of 'GrowthModelComponent'. They do

⁸Another would be polymorphism through inheritance or interfaces

⁹I have chosen the 'Tree' to be composed of 'TreeCrown' and 'Stem' because foresters are interested in the stem after the tree has died. Aggregation does not allow for this independence.

Example 3.3.1 Forwarding of the message 'grow()' in 'GrowthModelComposite'.

```
...
private vector children;
...
public void grow(){
    Enumeration e = children.elements();

    while(e.hasMoreElements()){
        GrowthModelComponent c = (GrowthModelComponent)e.nextElement();
        c.grow();
    }
}
```

not forward any messages. The class 'TreeCrown' inherits from 'GrowthModelComposite' and thus forwards messages to its sons, objects of the type 'Branch' and 'Leaf', by default. Hence, the behavior of an object of type 'TreeCrown' is totally determined by the behavior of its sons. If you want to add behavior to the aggregated object, in this case 'TreeCrown', one possibility is to override the inherited methods or to inherit directly from 'GrowthModelComponent', as class 'Tree' does, and implement your own choice of behavior and forwarding (see figure 3.4 on page 41).

Clients who want to interact with objects in the aggregate¹⁰ use the 'GrowthModelComponent' interface. Applying the pattern 'Composite' has the following consequences (Gamma et al., 1995, p. 166):

- Hierarchies of primitive objects and aggregated objects can be defined. Primitive objects can be aggregated to complex objects. Client code written for a primitive object can also work on aggregated objects.
- It simplifies the client. A client can send the same messages to individual objects and aggregates. This eliminates most cases which had to be handled by constructs such as 'switch'.
- The client does not need to be modified with the introduction of new 'GrowthModelComponents'.

¹⁰It is common to ignore the semantic difference between composition and aggregation, though incorrect.

- The mechanism for adding, removing and accessing sons allows the composition of unwanted 'GrowthModelComponent's to a 'GrowthModelComposite'. The type checking system of the compiler does not object to the addition of two 'Stem' objects to a 'Tree' object. These kind of checks have to be implemented and take place during execution time.

There are other disadvantages which surface during implementation. Every object of type 'GrowthModelComponent' has to implement or inherit the methods for administrating sons even if they do not have any. In case an unwary programmer uses these methods on an object like that, it cannot behave as expected. The programmer has made a semantic error which is not detected by the compiler. Hence, an exception should be thrown.

Figure 3.5 on page 42 shows a solution to this problem. The type 'Composite' is abstracted to an own interface, or possibly class ¹¹.

In the composite pattern as presented by Gamma et al. (1995, p. 163) there are three players: components, composites and leaves. My solution allows the leaves not to inherit the component methods. Clients access the objects via the 'GrowthModelObject' interface. They never cared to know about the composite character of the objects anyway. Objects of the type 'GrowthModelComposite' can be produced by inheriting both from 'GrowthModelObject' and 'Composite'. One even might create such a class to provide implementation inheritance. The messages, which are forwarded down the composite tree, belong to the interface 'GrowthModelObject' in both cases. If a programmer tries to add an object of type 'GrowthModelObject' to a leaf player, the compiler is going to notice. A class taking on the role of a leaf might inherit or implement the component interface later on, if needed and useful, providing the same extensibility.

¹¹I use an interface to avoid problems with multiple inheritance which some languages, such as Java, do not provide.

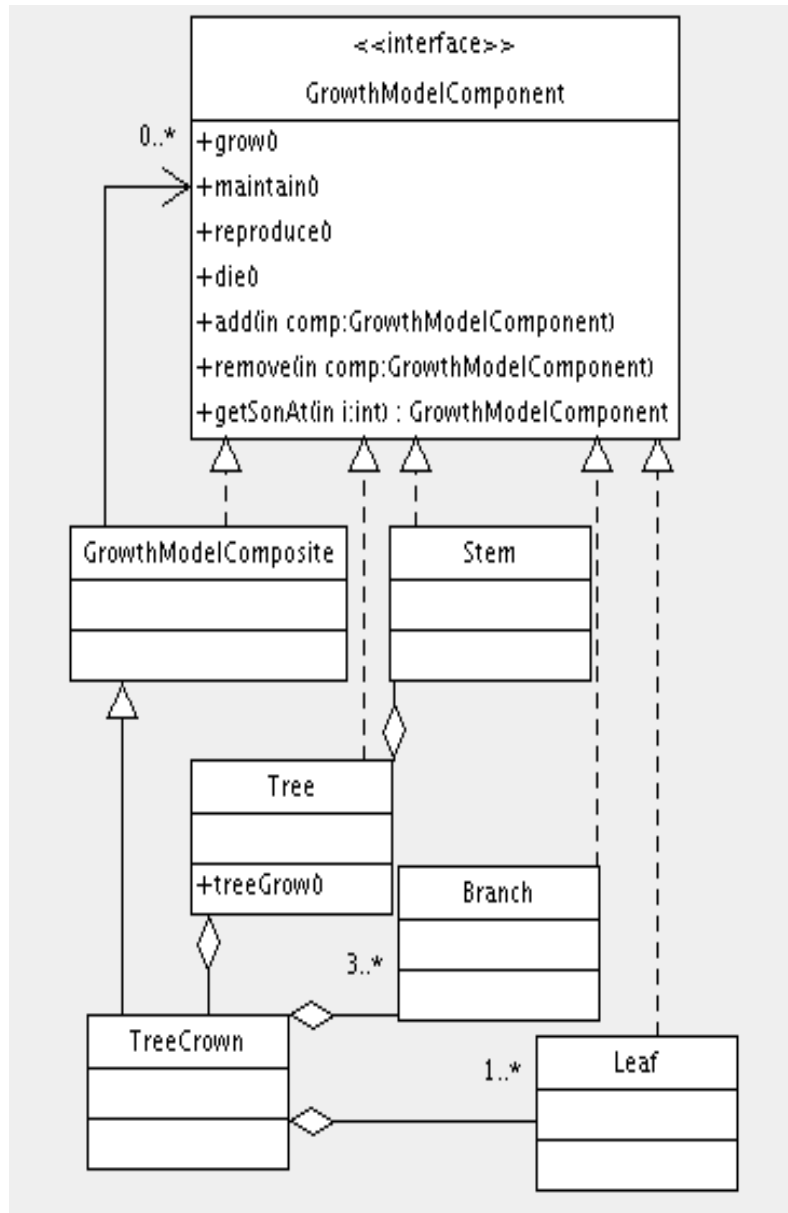


Figure 3.3: Composite pattern.

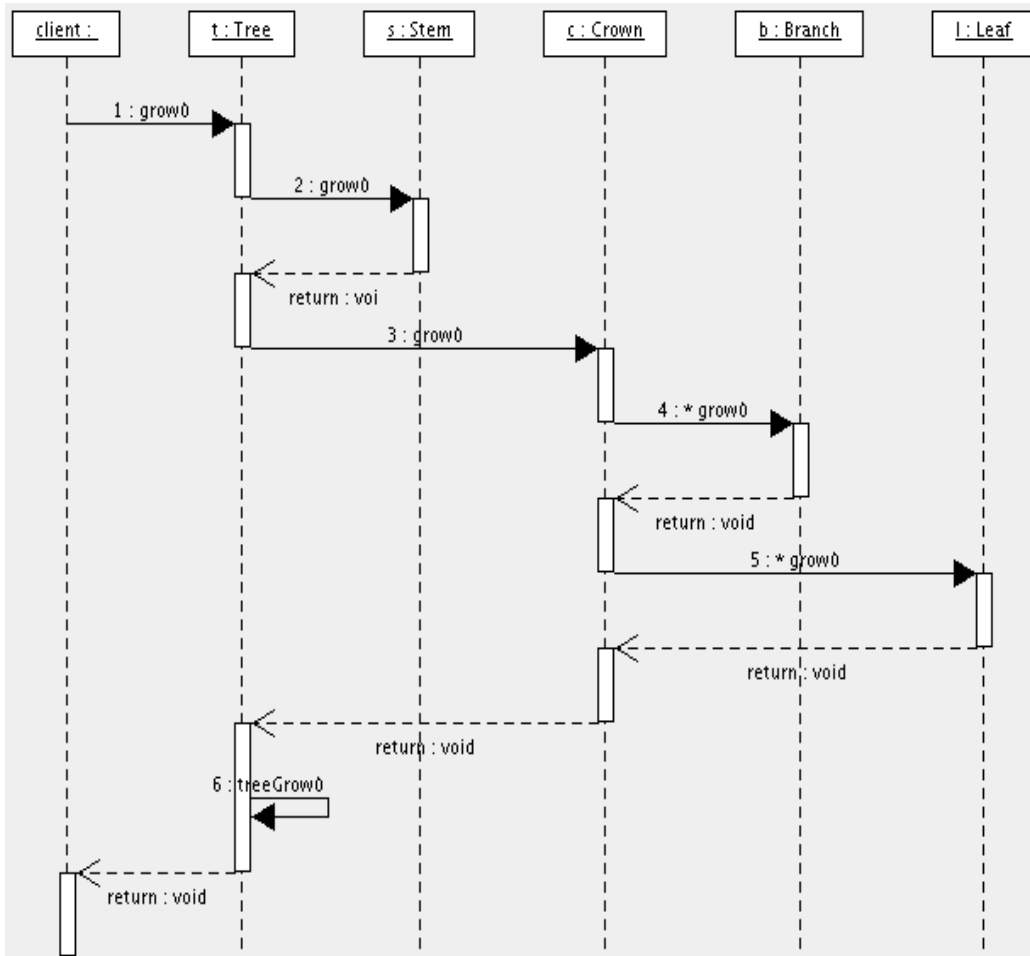


Figure 3.4: Sample message sequence for example 3.3 on the preceding page.

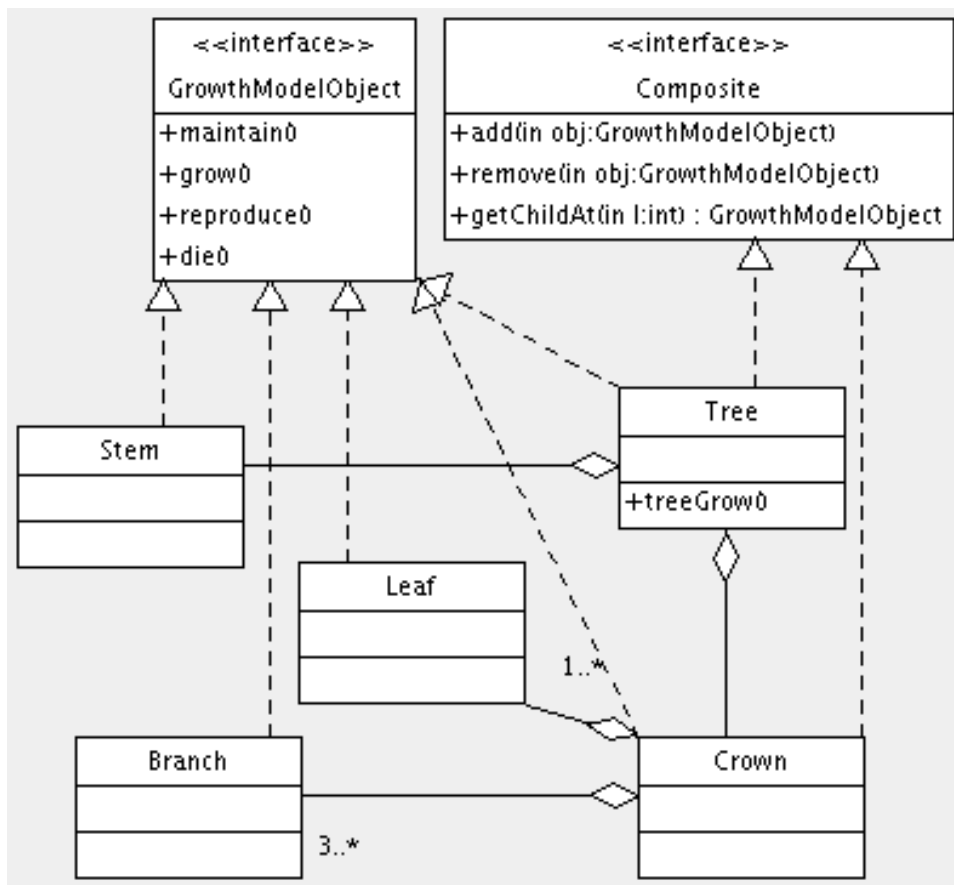


Figure 3.5: Modified composite pattern.

3.3.2 Template Method

The template pattern is useful in factorizing the behavior of classes by localizing it in a common base class. Thus, it helps to avoid code duplication (Gamma et al., 1995, p. 325). It can be used if the structure of an algorithm is the same for a class of objects while some implementation details vary among subtypes.

For example, the lifeprocess of a tree can be seen as composed of maintenance, growth, reproduction and death as shown in use case¹² 2.4 on page 17. Although maintenance, growth, reproduction and death differ from species to species, they are part of the life-process of all 'LivingObject's. Furthermore, the subparts of the 'live-algorithm' can be seen differently depending on the model.

The template pattern is applied by creating an abstract class 'LivingObject' which declares the methods void live(), LifeUnit maintain(), reproduce(), grow() and boolean die(). 'LivingObject' might look as in example 3.3.2 on the following page. The methods 'reproduce()' and 'grow()' reduce the value of variable u as a side effect¹³.

The template method 'live()' defines the skeleton of the algorithm. It is declared final to prevent its modification in subclasses. The methods 'maintain()', 'grow()', 'reproduce()' and 'allOver()' are primitive operations which must be defined appropriately by subclasses. They are declared protected to keep them from being called by methods other than the template method.

The method 'doSomethingElse()' is called at the end of the 'live' method. It is a hook operation. Hook operations may or may not provide default behavior. Furthermore, they are not abstract and therefore do not need to be redefined in subclasses¹⁴.

The template method 'live()' calls methods which are defined in subclasses, thereby inverting the usual control structure. Furthermore, the subclasses of 'LivingObject' can add new methods. The method 'die()' is provided with a default behavior which is useful in all subclasses. It may but need not be overridden in a subclass.

The template pattern is shown here because it is likely to be useful in the implementation of a particular forest growth and yield model or model family.

¹²There are, of course, numerous ways to model the life-process.

¹³Maybe not the most elegant solution but sufficient as an example.

¹⁴It must be clearly documented which methods may and which must be redefined.

Example 3.3.2 Sample live() method in 'LivingObject'.

```
public abstract class LivingObject{
    private Vector myPals;
    private boolean isDead;
    ...
    public final void live(){
        // maintain
        LifeUnit u = maintain();
        if(u > MINIMUM
            && age > REPRODUCTIVE_AGE){
            try{
                reproduce();
            } catch(PartnerNotFoundException e){};
            if(u > GROWTHMIN){
                try{
                    grow();
                }catch(SpaceNotAvailableException e){}
            }
            if( allOver() == true ){
                die();
            }
            age++;
            doSomethingElse();
        }

        protected abstract LifeUnit maintain();
        protected abstract void grow()throws SpaceNotAvailableException;
        protected abstract void reproduce()throws PartnerNotFoundException;
        protected abstract boolean allOver();
        protected void die(){
            isDead=true;
            // Spread the news!
            Enumeration e = myPals.elements();

            while(e.hasMoreElements()){
                Pal c = (Pal)e.nextElement();
                c.notify();
            }
        }
        protected abstract doSomethingElse(){};
    }
}
```

Nevertheless, the implementation effort should be minimized. This can be achieved by making the number of primitive operations which need to be implemented as small as possible. Hook operations are somewhat less problematic especially if they provide a default behavior. Hence, they need not be redefined often.

Appendix A

Terms & Abbreviations

Abstract Factory: Provide an interface for creating families of related or dependent objects without specifying their concrete class (Gamma et al., 1995).

ADT: An abstract data type (ADT) is defined by a set of operations and axioms, which specify the corresponding semantics (what happens to the state of the object when requesting it to do an operation)¹. Eiffel uses classes to realize ADTs. The attributes of an object are accessed indirectly by its operations.

Basal area: Surface area of an axial cut of a tree or stand at 1.3 meters height (Lemm, 1991, p. 196).

BHD: The BHD is the mean diameter of a tree stem at breastheight (Brusthoehendurchmesser). Breastheight is 1.3 meters above the ground (Lemm, 1991, p. 196).

Bridge: Decouple an abstraction from its implementation so that the two can vary independently (Gamma et al., 1995).

Class: Classes provide a generic specification of objects on an abstract level. This specification is provided at one place. Classes are what we deal with while programming. They play an important role in object oriented programming (Rechenberg, Pomberger et al., 1999):

¹Objects in OOP-programs do not have a 'free will', their behavior is well defined.

- They implement a type by their interface. This type can be used in declarations.
- A class defines a set whose elements are objects of the class.
- Classes provide structure similar to modules.
- Classes serve as a mechanism to create new objects.

d7: The d7 is the mean diameter of a tree stem at 7 meters above the ground (Lemm, 1991, p. 196).

Framework: Although a framework can provide code reuse through some of its classes it emphasizes the reuse of an application design (Gamma et al., 1995, p. 27). It does so by dictating the thread of control of the program. Hence, contrary to the toolkit, you write code which is called by the control flow fixed in the framework. Obviously, a toolkit leaves more freedom of design and therefore provides a higher amount of flexibility. The advantage of frameworks lies in reducing construction time by subclassing framework classes, specializing them to behave as required. By providing a fixed structure frameworks are easier to understand and maintain, look and behave similar.

Inheritance: Many classes are similar and differ only in details. Inheritance allows one class (the derived class) to define the behavior and data structure of its objects as a superset of the definition of another class (base class) or classes (Wirfs-Brock, 1990). The derived class (Rechenberg et al., 1999):

- can receive all messages that are understood by the base class.
- inherits a set of methods and variables of its base class, thus providing a mechanism for code reuse.
- may add variables and methods to its inherited behavior.
- may change its reaction to a message also understood by the base class, thereby changing inherited behavior.
- may not remove variables or methods inherited from the base class nor change their declaration in order to maintain compatibility.

Information hiding: The **behavior** of an object is a set of messages to which it responds. Not all of these methods need to be part of its publicly accessi-

ble interface (Wirfs-Brock, 1990). It may have other methods or attributes which it uses internally to perform a task. The object has a public interface which is open to other objects and a private representation which is kept quite distinct. Information hiding:

- removes some part of the things encapsulated by the object from view. Code can be more easily modified, maintained and extended.
- makes a distinction between the ability to perform a task and the specific steps needed to do so.

Lorey height Stand height weighted with the basal area (Lemm, 1991, p. 196).

Message: A message consists of the name of an operation and any required arguments (Wirfs-Brock, 1990). Objects can send messages to other objects that are known to them ². A message can cause a reaction in the receiving object (Rechenberg et al., 1999):

- It may cause it to tell the other object about it's state (values of variables).
- It may cause it to do one of it's actions.
- It may cause it to change it's state.
- It may cause it to do a combination of the above.

Object: An object is made of a set of attributes and a collection of operations working on those attributes. It has a state and behavior. The bundling of information and related operations is called **encapsulation**.

Polymorphism Objects of two or more classes may be able to respond to the same message (Wirfs-Brock, 1990). The response might be different because each object belongs to a different class but the message is understood. The classes are different, but similar, and responde to the message in their appropriate way. The sender can send a message without caring about the class of the receiver as long as the message can be received.

Toolkit The most common way of code reuse is to provide toolkits. A toolkit is a library of classes and class hierarchies which provide useful functionality

²In most cases, by reference.

(Gamma et al., 1995, p. 26). Examples include libraries of data structures and algorithms or a library for the construction of a graphical user interface. Developers can use the functionality of a toolkit, therefore sparing them the need to implement it themselves. Toolkit design is more difficult than designing an application, the designer has to avoid dependencies and prerequisites which unnecessarily restrict the toolkits applicability (Gamma et al., 1995, p. 26).

Top height: Mean height of the 100 trees with the largest diameter at breastheight of a stand per hectare (Lemm, 1991, p. 196).

Type: A type T consists of a set of messages. An object, which understands all messages $m \in T$ is of type T .

UML: The UNIFIED MODELING LANGUAGE (UML) is a formal language for objectoriented specification, visualization, construction and documentation of software systems (Erler and Ricken, 2002, p. 13).

Bibliography

- Erler, T. and Ricken, M.: 2002, *UML*, verlag moderne industrie buch AG Co. KG, Königswinterer Str. 418, 53227 Bonn.
- Fishwick, P. A.: 1996, Extending object-oriented design for physical modeling, *Submitted to ACM Transactions on Modeling and Computer Simulation*.
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison–Wesley, Reading, MA.
- Gregorius, H.-R.: 2002, *Modelle der Populationsdynamik*, Institut für Forstgenetik und Pflanzenzüchtung, Göttingen.
- Lemm, R.: 1991, *Ein dynamisches Forstbetriebs-Simulationsmodell*, PhD thesis, ETH Zürich, CH-8092 Zürich.
- Nagel, J., Albert, M. and Schmitt, M.: 2002, Das waldbauliche Prognose- und Entscheidungsmodell BWINPro 6.1, *Forst und Holz* **57 (15/16)**, 486–493.
- Pretzsch, H.: 2001, *Modellierung des Waldwachstums*, Parey Buchverlag im Blackwell Wissenschafts-Verlag GMBH, Kurfürstendamm 57, 10707 Berlin.
- Pretzsch, H. and Kahn, M.: 1998, *Forschungsvorhaben 'Konzeption und Konstruktion von Wuchs- und Prognosemodellen für Mischbestände in Bayern': Abschlußbericht Projekt W28 Teil 2. Konzeption und Konstruktion des Wachstumsmodells SILVA 2.2 – Methodische Grundlagen*, Lehrstuhl für Waldwachstumskunde der Ludwig-Maximilians-Universität München, Freising.

Rechenberg, P., Pomberger, G. et al.: 1999, *Informatik Handbuch*, Carl Hanser Verlag, München, Wien.

Wirfs-Brock, R.: 1990, *Designing Object Oriented Software*, Prentice-Hall, Inc., Eaglewood Cliffs, New Jersey 07632.