Softw Syst Model (2003) 2: 5–14 / Digital Object Identifier (DOI) 10.1007/s10270-003-0020-3

Models in software engineering – an introduction

Jochen Ludewig

Institute of Software Technology at Stuttgart University, Breitwiesenstr. 20–22, 70565 Stuttgart, Germany;
E-mail: ludewig@informatik.uni-stuttgart.de

Received: 21 October 2002/Accepted: 10 January 2003
Published online: 27 February 2003 –

Abstract. Modelling is a concept fundamental for software engineering. In this paper, the word is defined and discussed from various perspectives. The most important types of models are presented, and examples are given.

Models are very useful, but sometimes also dangerous, in particular to those who use them unconsciously. Such problems are shown. Finally, the role of models in software engineering research is discussed.

Keywords: Models – Software engineering – Metaphors – SESAM

1 Disclaimer and goals

We use models when we think about problems, and when we talk to each other, and when we construct mechanisms, and when we try to understand phenomena, and when we teach. In short, we use models all the time.

That means: models have never been invented, they have been around (at least) since humans started to exist. Therefore, nobody can just define what a model is, and expect that other people will accept this definition; endless discussions have proven that there is no consistent common understanding of models.

In this paper, this difficulty is (almost) ignored, and the term "model" is defined as if there were no conflicting opinions. This approach is taken because it is the only way (at least the only way known to the humble author) for investigating the power and the limitation of models. Readers are not required to accept my definitions permanently; but they might be prepared to accept them at least while they read this contribution, because my judgements and conclusions are based on my definitions.

This paper is intended to help the reader

- recognize models where they appear,
- know the properties and the power of models,
- clearly distinguish models from the original objects,
- create new models where appropriate.

2 The air we breathe

Our ability for modelling is not acquired but given to us from birth. Without it, we would not be able to reduce the vast flow of information to a rate we can cope with. By mapping visible and invisible phenomena to *notions* (in German: *Begriffe*), the number of *different* observations is significantly reduced, and we deal with some classes of problems rather than with millions of individual problems. Hence, we can collect experiences, find generic solutions and decisions, and develop strategies for surviving in the real world. The ability for reflection, which is considered *the* difference between man and animal, is directly related to the use of models.

The particular strength of models is based on the idea of *abstraction*: a model is usually not related to one particular object or phenomenon only, but to many, possibly to an unlimited number of them, it is related to a *class*. They who note that the change from high tide to low tide and from low tide to high tide follows a certain rhythm can prepare for, or make use of it. Those who learn that a certain class of animals rather than one single living creature is fast, strong, and dangerous, have improved their chances for survival.

While we live, i.e. act and react, we use models all the time, usually unconsciously. The situation is quite different in research and engineering: there, the creation of models is an explicit topic; it is the purpose of research and an important step in producing artefacts. Research yields theories. A theory is a special kind of model (see below). The more influential a theory is in the world, the higher it is estimated. Such influence ranges from a change of our perception of the world (like switching

from a geocentric to a heliocentric view) to a massive effect and threat like that of nuclear bombs.

The role of modelling in engineering is similar. Models help in developing artefacts by providing information about the consequences of building those artefacts before they are actually made. Such information may be highly formal (like the theory of mechanics, as it is applied in the construction of bridges, or the theory of computational complexity which is used in the analysis of algorithms) or rather informal (like a rough drawing of a machine, or a textual specification of a software system). Interface definitions are models too; they are particularly important for the spreading of technology.

3 Definitions

The term "model" is derived from the Latin word *modulus*, which means *measure, rule, pattern, example to be followed*. Obvious examples are toy railways and dolls, maps, architectural models of buildings. In software engineering, we have process models, design patterns, class diagrams. Other models are less obvious, like project plans, specifications and designs, metrics, and minutes of project meetings.

3.1 The model criteria

In order to distinguish models from other artefacts, we need *criteria*. According to Stachowiak [8], any candidate must meet three criteria, or otherwise it is not a model:

- **Mapping criterion**: there is an original object or phenomenon that is mapped to the model. In the sequel, this original object or phenomenon is referred to as "the original".
- **Reduction criterion**: not all the properties of the original are mapped on to the model, but the model is somehow reduced. On the other hand, the model must mirror at least *some* properties of the original.
- **Pragmatic criterion**: the model can replace the original for some purpose, i.e. the model is useful.

The mapping criterion does not imply the actual existence of the original; it may be planned, suspected, or fictitious. The dwarfs and fairies found in many gardens model fictitious creatures, and a map of Troy mirrors a historic theory that is not at all generally accepted. Most novels and movies model a reality that was born in the author's fantasy. The cost estimation of a software project is a speculative model of the future.

A model may act as the original of another model, we can find cascades of models, e.g. when a painting shows a room with paintings. A program design is a model of the code to be written, while the code is a model of the computation performed by the computer when the code is executed.

At first glance, the reduction criterion seems to describe a weakness of models, because something is lost in the model that was present in the original. But that loss is the real strength of models: very often, the model can be handled while the original cannot. A map is handy and cheap. Even when we can use a space-ship (which is neither handy nor cheap), we cannot identify most of the details that are clearly marked in the map.

The pragmatic criterion is the reason why we use models. Since we are not able or not willing to use the original, we use the model instead. That applies to a toy that represents an extremely friendly animal, and it also applies to the theory of a big bang as the birthday of our universe, because no scientist can directly observe what happened billions of years ago.

Figure 1 shows the relationship between original and model. Note that a model is not necessarily similar to the original in any naive sense, like a toy automobile that looks similar to a real automobile. The attributes may be mapped in many different ways. In photography, colours are translated into values on a grey scale. Physical properties may be recorded as numbers. Artists express feelings by shapes and sounds; the daytime is indicated by the angles of two clock hands.

As an effect of the reduction, many features of the original (the *waived attributes*) are not found in the model. For example, the name of a person is not visible in his photograph. On the other hand, features that do not stem from the original are added (*abundant attributes*). For example, the size of the picture does not tell anything about the person. A Z specification does not comprise most of the details later found in the program, but many syntactical details (like arrows etc.) that are given as part of the specification language; their shapes have no meaning for the system to be developed.

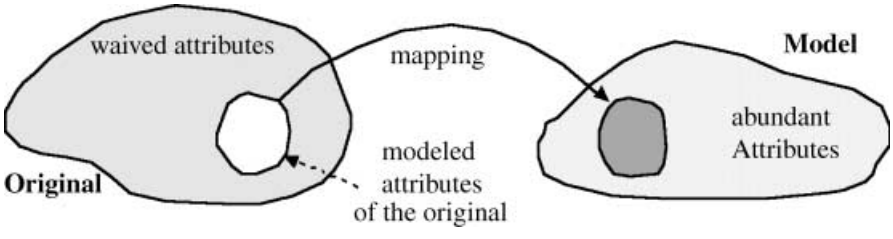

Fig. 1. Original and model according to Stachowiak

Let us use an identity card as an example:

The IC is issued for a particular (say female) person (mapping criterion). If the card is faked, that person does not really exist. Most of her properties are waived, like her memories, her taste, or her favourite destination for holidays (reduction criterion). A number gives her height; her residence is indicated by her address. The IC is useful (pragmatic criterion) when somebody wants to find out certain properties of the person who might not be able to answer questions (e.g. after an accident) or not be willing to answer them correctly (e.g. when the person tries to open a banking account).

3.2 Related terms

Many frequently used terms have a meaning that is similar to the meaning of the term "model" (see Fig. 2). Let us discuss which of those terms can be regarded as proper models. There is, of course, no sharp distinction.

Tool: many tools, perhaps all tools have been invented as imitations of existing aids. A hammer is an improved copy of the human fist. But a tool is not a model: The relationship to its original very soon becomes obsolete, because the tool is modified again and again, independently from its original. In the early days of computing machines (i.e. two hundred years ago), those machines copied the notes people write on paper when they calculate. Modern computer hardware no longer behaves in any way similar to what people do; a computer is *not* an artificial brain.

Name: a name is not a model; while it can be used in place of the object it identifies, it does not provide any information about the object.

Icons represent programs and files. Any action on an icon is a substitute for an action on the program or file. But little, if any, information about the original object is contained in the icon. Therefore, icons are not regarded as models. The same is true for **symbols**.

Metaphor: describing a complicated machine, we often identify a central part, and refer to it as the *heart* of the machine. When a term from one area (e.g. from biology) is used in a different area (like mechanical machines) in order to describe some characteristics (it keeps the whole thing running) it is called a metaphor.

What about the model criteria? Let us look at the term "software virus" as an example. First, there is an original (the harmful code; note that the biological virus is *not* the original; it only provides the *word* and its *connotation*). Second, the metaphor is an extremely reduced representation of the original. (When somebody tells us about a new virus, the metaphor conveys some information about the type of program, but not about its size or its algorithm). Third, the original is represented by the metaphor, and we can use the metaphor when we think about defence against it. (A virus is dangerous, and may be passed to another victim unintentionally. We can prevent an *epidemic* spread of a virus by keeping the *infected* computer in *isolation*.) A metaphor *is* a special type of model.

Metaphors are particularly useful when we deal with something that is quite new, and not yet named. Computers, including all their details and attributes, were invented only recently, and there was a complete lack of genuine names. Few, if any, new words were introduced; the vast majority of missing names were replaced by metaphors. Therefore, we use the terms *jump*, *loop*, *crash*, *freeze*, *bug*, *overflow, storage* and *memory*, *maintenance*, *firewall*, *protocol*, *message*, *window* etc. etc. Most of these metaphors were presumably created spontaneously, only a few were (probably) invented (like file, desktop, transaction, handshake).

Some metaphors are misleading. A typical example is *inheritance*, as used in object-oriented programming. In its traditional meaning, inheritance applies to individuals: when the parents die, their property is passed on to their children. In biology, the word was used in a similar meaning: parents pass their genetic material to their children (but the material is copied). The meaning of inheritance in object-oriented programming is very different: it applies to classes of objects rather than to objects. And it establishes a permanent dependency: when a class is modified, all dependent classes are implicitly modified too. This may be the reason why so many programmers fail to understand inheritance.

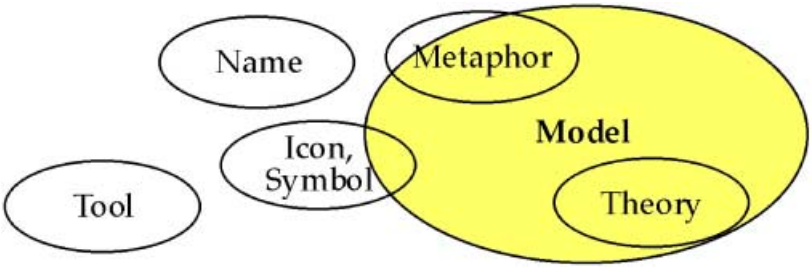

Fig. 2. "Model" and related terms

Another example is *maintenance*. While the word implies a conservative approach (maintain the original state), software maintenance is done for the sole purpose of changing the state. The meaning has been inverted!

Theory: a theory widely used in computer science is the theory of finite state machines. The finite state machine is a model of the computers we use, or of some parts of our computers. The theory of finite state machines is a model of the behaviour of our computers. The fact that switching takes some time in any physical device (i.e. there are states between the states) is ignored. By applying this theory, we are able to predict the behaviour, and we can construct useful machines. In general, every theory is a model of some phenomena that can be observed and/or induced. A theory is a highly abstract type of model; it emphasizes results and conclusions rather than obviousness.

Note that theories need not to be formal. Brook's law ("Adding manpower to a late project makes it even later") is an example of an informal theory.

4 Model taxonomy

4.1 Descriptive and prescriptive models

Models in a narrower sense (i.e. not including metaphors) can be classified under various aspects. A model can mirror an existing original (like a photograph), or it can be used as a specification of something to be created (like a construction plan). In the former case, we call it a *descriptive* model; in the latter case, we call it *prescriptive*. When an architect sketches an old house, and then adds to his drawing some modifications he suggests to the owner, the model is first descriptive, later on prescriptive. We call it a *transient* model.

Though we talk about descriptive or prescriptive models, this is in fact a property of the *relationship* between a (particular) model and a (particular) original rather than a property of the model. A construction plan is not necessarily prescriptive; it may have been produced from an existing object. Therefore, a model may be descriptive with respect to one original, and, at the same time, be prescriptive with respect to another original. A simple example is a drawing of an antique golden ring that is used for reproducing similar rings.

At first glance, it seems obvious that descriptive models can only be created after the original, while prescriptive models need to exist before the original is made. The latter is in fact true, but the former is not (or not precisely) true: prognostic models that describe something that does not yet exist are descriptive, because they are not intended to influence the original. The weather forecast is such a model: though we cannot (yet) influence the weather consciously, we can fairly reliably describe the weather we will see tomorrow. The same is true for a cost estimation, which has no (direct) influence on the actual cost. Such estimation should be clearly distinguished from a requirement ("The cost of the project must not exceed 500 k€"), which is, of course, a prescriptive model. If there is more than one prescriptive model, those models may be inconsistent. There may be a requirements specification that turns out to be inconsistent with the cost limit; then, there is no solution that fits both models.

Descriptive models are applied in order to make some specific information about the original easily and quickly accessible: the table of contents in a documentation reduces the time for finding a specific topic, and the program size (in LOC, e.g.) can be used for simple conclusions, like "if program A is ten times larger than program B, it will probably not fit into the available memory."

4.2 Purposes of models

The purpose of the models is another criterion that can be used for classifying models.

- *Documentation* is created when data in its most general sense is derived from data that is already existing and available. Therefore, it is descriptive. Three important subclasses are
  - *concise descriptions* as stored in software databases or included in tenders and marketing information
  - *minutes, protocols*, like the log of a test, or of the dialogues in which the customer describes his or her requirements. The requirements specification (which is prescriptive) mirrors those requirements.
  - *metrics*, both *software metrics* and *process metrics*, are highly abstract models. The number of pages of a specification, the number of classes in an object-oriented program, the response time of a program, and the memory it occupies are examples of software (or product) metrics. The duration and cost of development and the number of developers are two important process metrics.
- *Instructions* (like "copy the file to your harddisc") provide information about some activity (like installation, or test). Instructions are prescriptive.
- *Explorative models* are transient (see Fig. 3). They are used when the consequences of a change are to be evaluated. The modifications are applied to the model rather than to the real system. When their effect seems to be positive, the modifications are applied to the original. If we are not sure about a new interface of an information system, we implement a prototype that can be evaluated, even though it does not provide the functionality of the target system.
- *Educational models* and *games* replace the originals for ethical or practical reasons. Examples are models of the human body as used in medical education, flight simulators, and the dolls children play with. All these

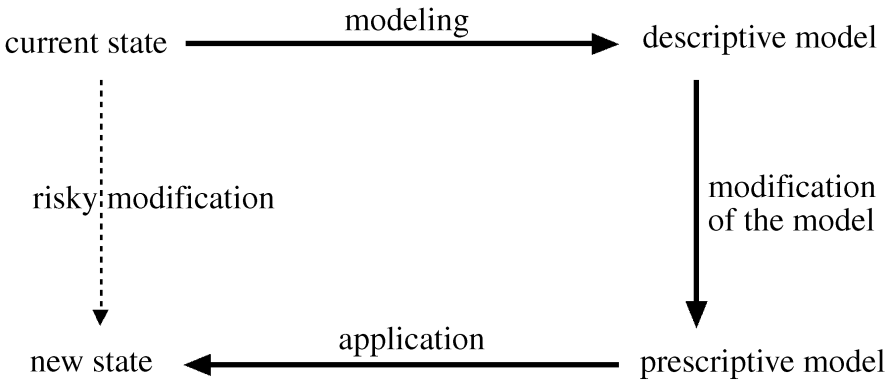

Fig. 3. Application of an explorative (i.e. transient) model

(descriptive) models somehow imitate the appearance of the originals. This is also true for non-material models of some objects that may be distant or non existing ("*virtual reality*").
- *Formal (mathematical) models* (e.g. the formulas used in physics and chemistry) are descriptive, too. Unlike educational models, formal models do not resemble the reality they describe. They allow us to analyse or to forecast phenomena of the real world. Other more complex models like a flight simulator often contain such mathematical models.

5 Models in software engineering

Models can be found in all areas and applications of software engineering. Some of them are discussed below.

While software developers create concrete models (see below), people who do research in software engineering work on notations and methods for developing such concrete models. Finite state machines and state charts, Petri nets and data flow diagrams are a few examples of models that use such notations.

5.1 Prescriptive models for software engineering

Most of the models used in software engineering are prescriptive, for instance:

- *process models*, like Cleanroom Development, or Extreme Programming
- *information flow models* like the diagrams used in SADT
- *design models*, like class diagrams, or boxes representing the module structure
- *models of user interaction*, like use cases, or interaction diagrams
- *models of principles used for constructional details*, like design patterns
- *process maturity models*, like CMM or SPICE (which are actually sets of models). In these cases, each model implies a criterion for judging existing processes.

5.2 The document chain

When software is developed in the traditional waterfall approach, documents are generated each of which is prescriptive for the next one. Only the requirements specification is double-sided, because it describes the user's needs, and it prescribes the product to be developed (see Fig. 4). It is this double role that makes the specification the most important software component. The chain runs from the specification to the architectural design, from there to detailed design, code, and finally execution. User's manual and test data are descriptive models of the specification; they can replace the specification for certain purposes.

One of the hard, basically still unsolved problems of software engineering is the difficulty of maintaining the logical chain of documents when any of the documents is modified. The identification and subsequent modification of other components in order to keep the whole system consistent is called *tracing*. When the tracing follows the sequence in which the documents have been (or should be) produced it is called forward tracing. When the origin of change is in one of the late documents (e.g. in the code), we need backward tracing. The problem is extremely hard because in every step of the sequence some information is lost, while some other information is added. This is why there is no automatic correction of related documents.

5.3 Software as a model of the world

> *"In many systems the machine embodies a model or a simulation of some part of the world. (...) The purpose of such a model is to provide efficient and convenient access to information about the world. By capturing states and events of the world and using them to build and maintain the model we provide ourselves with a stored information asset that we can exploit later when information is needed but would be harder or more expensive to acquire directly."* (M. Jackson in [7]).

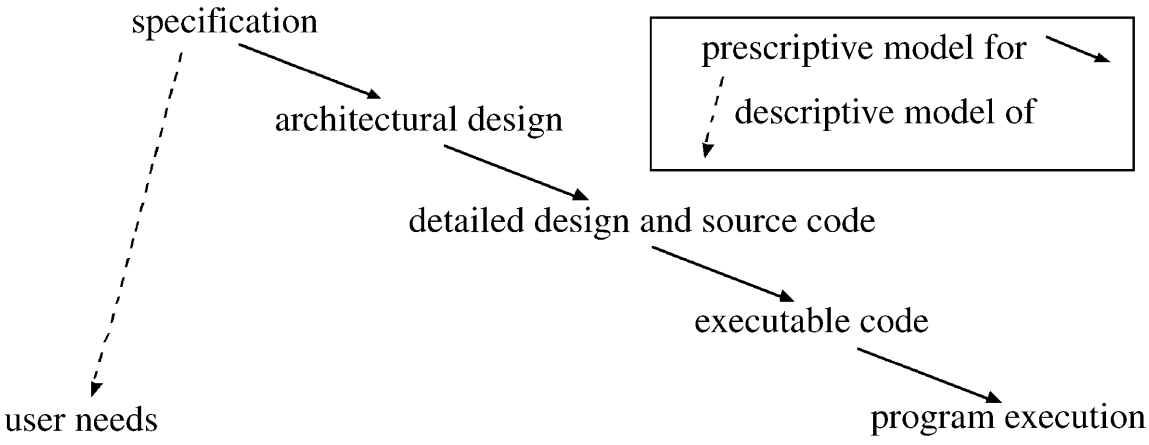

Fig. 4. The (simplified) document chain; some documents (like the user's manual, e.g.) and some arrows bypassing some of the documents are not shown in Fig. 4

One of the popular statements in software engineering says that many (or even all) software systems represent some part of the world. Is that true, or is it a mere conjecture?

Traditional (i.e. electrical and mechanical) engineers usually do not create models; their artefacts are tools that do not model anything. Neither a bicycle nor a radio or a house is a model of something; they are means that enable us to satisfy our needs better, and with less effort than without them. And many achievements can only be thought of when technical solutions are available: nobody could possibly walk to the moon, or watch bacteria at work. Technique enables and provides possibilities, usually without models. The symbol of engineering, the wheel, is an invention that does not model anything.

While software systems in general serve the same purpose as other technical solutions, i.e. help us in achieving something, they are not only described in terms of the real world, but they often actually represent it. Explaining a software system, we tend to say things like "In this module, the customers are stored, and this procedure will decide whether or not their orders are accepted." The computer system seems to be a doll's house, a mirror of the world outside the computer.

Why is that so? What is so special about our systems?

Our emphasis on models has good reasons:

From the very beginning, man as a thinking and deciding creature with a large memory has been the ideal for general-purpose computers. Computers (and hence computer software) imitate man. Centuries ago, people dreamed of machines that could generate music and play chess, and a chess playing machine was in the mind of Konrad Zuse as early as 1937, when he built his first computer. (He even expected that fifty years later, probably a machine would play better chess than a human; that was very close to reality!) That means: computers were designed to mimic (i.e. model) human thinking, even though we do not really understand how our brain actually works. Most of the tasks we give to the computers have traditionally been human tasks, and we simply use the models we have traditionally in mind. There have e.g. been accounting systems ages ago, and there is no reason to change our models when a machine does the job.

When we produce new software, our fantasy and creativity are insufficient for creating something that is *really* new, because we are bound to the world we know, and we cannot invent something that we cannot imagine. The world of software is completely artificial, and it is for our soul as comfortable as a box made from stainless steel would be for a bird. Therefore, we *need* to rely on notions and ideas from outside the box, i.e. on models of the (mostly physical) world we live in. In object oriented programming, that approach was made one of the fundamental concepts, but it is not that new: in JSD (Jackson System Development [1]), we start by modelling the real world, before we introduce mechanisms which can later be implemented by machines.

Modelling the world in corresponding structures has also another advantage that Jackson mentioned in the seventies already: since the world is subject to change, the software must be changed too. Changes in the world are usually consistent with the existing structures. When the laws for the tax on cars are modified, some software systems used for computing that tax have to be modified as well. If entities like cars and owners exist in the software, such changes are comparatively easy. If they do not exist (say there is only an entity "taxable object"), then the changes are very hard to implement.

However, the fact that we only build software systems that imitate the existing world severely limits our possibilities. It is precisely the departure from existing models that marks the breakthroughs that many engineers have achieved. In the early days of combustion engines, cars looked like carriages; but that turned out to be a useless limitation. Using high frequency radio waves for communication is very different from any traditional way of communicating; that is why it is far more powerful. In some limited areas, the same principle has been applied

to software systems. Quicksort and Heapsort, two sorting algorithms whose performance is far better than that of traditional sorting algorithms, are based on principles that nobody would apply when sorting playing cards or papers in a file. Very few people can actually understand *how* a Fast Fourier Transformation works, but everybody is surprised to see *how well* it works. In all of these examples, the problem to be solved can easily be described in mathematical notation, without any fuzziness and ambiguities, allowing for the application of formal methods. It is hard to say whether or not we will some day be able to find similar approaches for problems that are really complex and hard.

When the first power lines were built, engineers would try to have a straight cable from source to sink in order to avoid losses due to frequent changes of the direction of flow. They apparently had in mind some flow of material. That model was no longer necessary when the theory of electricity was sufficiently well understood. In software engineering, we are not nearly at that point.

5.4 Metaphors for users and developers

> *Each XP software project is guided by a single overarching metaphor. (...) Sometimes the metaphor needs a little explanation, like saying the computer should appear as a desktop, or that pension calculation is like a spreadsheet. (...) As development proceeds and the metaphor matures, the whole team will find new inspiration from examining the metaphor. (...) The metaphor in XP replaces much of what other people call "architecture."* Kent Beck [3, p. 56]

In XP (Extreme Programming), metaphors play an important role. Kent Beck seems to be sure that software architecture and user interfaces are congruent. If it looks like a desktop (from a user's perspective), the developer should think of it as a desktop, too.

This concept oversimplifies the situation. The dustbin on the screen is not architecture; it is a simple representation for a highly complicated device. The metaphor is weak: it simply represents the fact that we can get rid of something by moving it into the dust bin, and we can retrieve it as long as the bin has not been cleared. Everything else is wrong: the electronic bin is often empty, but never full. Any file, even a very large one, fits into the bin. The bin contains files that are well organized, and do not have any influence on each other: files retrieved from garbage do not smell.

Users, in particular beginners, need metaphors in order to master their difficulties. In many cases, the metaphor is the only explanation the user gets. (Did anybody ever read the specs of a desktop dustbin?) The metaphor should be "watertight", i.e. the system should not corrupt it by inconsistencies. The happy few who use Macintosh computers must move a disc into the garbage not in order to destroy it but in order to release it from the computer. (This is not nearly as bad as the fact that the unlucky many have to click on "start" in order to shut down their systems, which resembles the idea to switch off a motor bike by using its kick starter.) When I send a letter, I want it to be delivered in due time and in good shape; if the address is wrong, the letter should return to me. These expectations hold for electronic mails as well. But sometimes we receive an error message, though the message *did* arrive at its destination.

A model for the user does not imply any model for the software architecture. In most cases, the metaphor is faked, i.e. its outside appearance is not at all similar to its inside structure. The architect can use the metaphor as a specification (of the outside appearance), but not as a hint for the architecture. When an engineer designs an aeroplane whose rudders are controlled electronically ("fly by wire"), there is no need to imitate any mechanical gear. The architecture is very different from an equivalent mechanical solution.

6 Risks from using models

> *A message to mapmakers: highways are not painted red, rivers don't have county lines running down the middle, and you can't see contour lines on a mountain.* William Kent [2]

Good models can replace the original very well. Therefore, good models tend to be confused with the original; very often the user does not even notice that there is such a thing like a model. For instance, most people believe that they can actually see a text file on the screen. They rarely think about all the problems related to the difference between a file and its representation. And they believe that all the details they see on the screen are the actual details of their files. Other, more threatening examples follow below.

6.1 A distorted view on the world

> *The world is simply object oriented.*
> A professor of informatics, 1992

The French word *deformation professionelle* describes the distorted picture many people have due to their professional attitudes. A medical man tends to classify the people he knows according to their diseases, sometimes even in private life. A teacher of logic tells his students that any statement is either true or false; if he applies this wisdom to the education of his children, he will probably fail. And many software people are so happy about the power of object-oriented programming that they really believe in an object-oriented world. When they are hit by counterexamples, they will not hesitate to explain to us that the reality is wrong, but their view is right.

That is ridiculous. One of the most important achievements of modern (i.e. 16th century) physics was the perception that we know nothing but models. In order to improve our models (i.e. in order to make them more useful), we have to compare the reality and our models again and again, and if they do not agree, the reality is always right and the model is always wrong. Those who believe in their models are not scientists but missionaries. Software engineering circles are full of them.

6.2 Examples from everyday life

The effects described above are frequently found, but usually less obvious. Software people call this "garbage in, garbage out".

If an analyst compiles the requirements in a requirements specification, her result is, as mentioned above, a model of the requirements she was given. Very often, the people she talked to were not extremely competent, and they were not able to check the written document against their real expectations. The developer will nevertheless take this document as the real and complete collection of requirements.

Words are magical: when we count branches, i.e. the if-statements in a piece of code, we get a number. There is nothing wrong with this number. But if somebody calls this number "the complexity of the program", he will usually forget very soon the primitive background of this measure, and will accept the number as a model of the complexity.

When people have tested a program for a while, identifying n bugs in the code, they tend to believe that they have found all the bugs. And they tell us that the program contained n bugs before testing. (Which implies the good news: now the program no longer contains any bugs.)

We all know: the program does contain more bugs. But there is at least a fair chance to find any particular bug by testing. Other deficiencies, like poor readability of the code, bad structure, lack of useful comments etc., cannot even be discovered by testing. When the results of the system test are accepted as the only indicators of software quality, the developers will inevitably try to optimise their work for this particular goal, producing unmaintainable code rather than sound software.

We can put the same statement in a more constructive way: in order to improve the situation in software engineering to achieve more reliable, robust, efficient, and maintainable software, we must develop models of software quality which reflect those properties. Next, we must teach the model and show its advantages. Finally, we must make sure that producing good software is not only beneficial for the organisation but also for the software developers.

Or even more down to earth:

- We need metrics far beyond those introduced by Halstead and McCabe which generate useful results that describe more than one very limited aspect.
- We should apply such metrics widely.
- We should not trust in people who insist on working without decent models, i.e. without decent plans, requirements, metrics, etc.

7 Descriptive versus prescriptive models in research

As stated in Sect. 2, creating models is the purpose of research. Scientists have provided lots of models since ages. Some of them (in Physics and Astronomy) date back to ancient times. All those models are descriptive.

Engineers need prescriptive models for building things, but their research produces descriptive models, too. We have excellent models of electric circuits, bridges, and mechanical devices. Software engineering seems to be an exception. Most results in this field offer new prescriptive models, like process models, or techniques for various activities. Is there any reason for this special position?

7.1 The difficulties of descriptive models

In order to construct a descriptive model, we need to know the original very well. In software engineering, the original is the real world of software projects, with requirements that are neither complete nor precise, with customers who tend to change their mind, with developers who suffer from insufficient education, with existing software that is very hard to modify, to name only the worst problems. In short: the real world of software projects is a mess.

In our group at the University of Stuttgart, we have been developing a system called SESAM (Software Engineering Simulation using Animated Models) since 1990 [5, 6]. SESAM is based on the idea that it should be possible to coach software project managers in the same way aircraft pilots are coached; SESAM is not a flight simulator but a software project simulator.

This simulator depends on adequate descriptive models of software projects and software project management dynamics. In order to discuss the modeling problems and challenges we experienced with SESAM, a short introduction on SESAM is necessary.

The user or "player" (who is supposed to be female in the sequel) takes the role of a software project manager; SESAM simulates the rest (customer, documents, process, employees). The player will start from an initial setting as a new project manager. The interactions between player and project are handled via keyboard and screen. The player receives messages, and enters her commands in order to make her project proceed. She can hire or fire employees and ask them to perform any of the tasks that are useful for software development (like start preparing a specification or revise the design document).

On the other hand, she receives messages about the things that happen in her project (for example, when

documents are completed or when an employee leaves the project). The time scale is compressed in order to cover a whole project in a couple of hours. When the game is over, the player receives her score, and some detailed analysis of her performance.

SESAM uses two models, one for the state of the project, the other one for the rules and relations that apply to software projects in general (Fig. 5). When running a game, the game state (which models the situation in one particular software project) is subject to continuous modifications. The rules and relations that determine these modifications constitute the so-called *project model*, which can be further divided into a static model and a dynamic one. While the *static model* defines the types and relations from which our virtual projects can be built, the *dynamic model* contains all the rules that represent the invisible mechanisms of a software project. Typical information contained in the static model is the set of document types that are developed in the project, and the fact that a programmer may read or write a document. A dynamic rule describes changes, like the effect of a review on the document that has been reviewed, or the effect of a meeting on the participants.

- The tutor must define the initial state of the project. This definition is transformed into an internal representation, the so-called *game state*. As simulated time proceeds, the game state (which models the situation in a real software project) is subject to continuous modifications.
- The rules and relations that determine these modifications constitute the so-called *project model* (or simply *model*), which can be further divided into the *static model* and the *dynamic model*. While the static model defines the types and relations from which our virtual projects can be built, the dynamic model contains all the rules that represent the invisible mechanisms of a software project. For better efficiency, the dynamic model is transformed into an internal representation (the executable model), which is interpreted by the simulator.

Typical information contained in the static model is the set of document types that are developed in the project, and the fact that a programmer may read or write a document. A dynamic rule describes changes, like the effect of a review on the document that has been reviewed, or the effect of a meeting on the participants. In the game state, the actual size of all documents and the number of errors in those documents is recorded, as well as the current motivation of the developers.

While developing and using SESAM we experienced typical problems arising with descriptive models.

The project model represents a theory of software projects that is based on empirical data. Such data is extremely hard to find; little has been published, and investigating such data in industry is practically impossible because those data have rarely been collected. Therefore, the model must be validated. But validation suffers from similar problems. There are no data to compare with. For the largest model so far [4], this validation has been done, though with a huge effort.

Other problems remain. Our model covers only a small fraction of the world; it is a compromise of simplicity and realism. Is it acceptable to ignore the private life of software developers? Which attributes of documents need to be modelled? Is it sufficient to count errors, or do we need different classes of errors?

And, last but not least, our models are not only limited due to technical reasons or in-sufficient knowledge; we had to learn that our students couldn't handle very complicated models. Therefore, we no longer try to

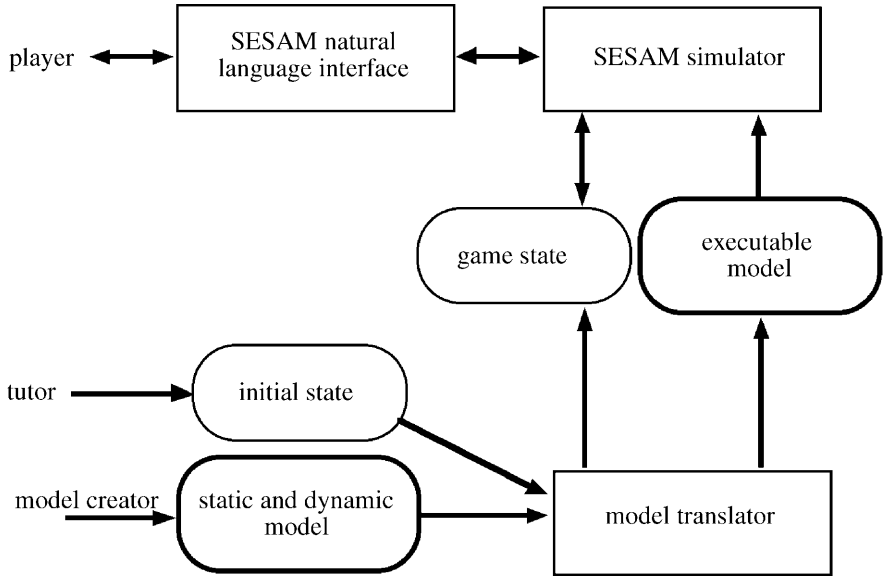

Fig. 5. SESAM architecture, simplified; models are shown in rounded boxes

add more detail to the models. Students should have a fair chance to succeed, because success is a reliable and cheap motivator. A good model is sufficiently detailed to be interesting, but sufficiently simple to allow for good results without weeks of training.

7.2 The dangerous charm of prescriptive models

While a descriptive model like SESAM has to represent its original very well (even when the player behaves stupidly), a prescriptive model can command whatever its author prefers. Their authors tend to take a position like "we do not care in what mess you are, we prefer to tell you what a wonderful world you could possibly live in."

That position seems to be reasonable provided the better world does really exist, and so does a path from the presence to that paradise. But most research projects have no chance at all to get to the point where they could possibly demonstrate that their ideas are applicable, or even superior. Most of them terminate when they have produced a doctoral dissertation and some printed paper, and their influence on the outside world remains very limited.

The problem here is not the fact that many of the ideas and concepts may be of little value; research has to produce a lot of garbage in order to yield a few excellent results. The problem is that most ideas are never tested. Scientific magazines, conferences, funding organizations, and university departments support the breeding of new ideas, but not their careful and time consuming evaluation and improvement.

Maybe there are too many models lacking the maturity that results from steady research. If more people contribute to our understanding of the existing world by analytical and empirical work, and evaluate what has been around for a while, we will get more useful results, and we will experience a steady increase of our knowledge, as expressed in generally accepted models.

But that requires a change of goals in research: Those who do not present new ideas, but compare and improve existing ones, should receive far more recognition.

8 Summary

In this article, I have hopefully demonstrated:

- Models are very important, in particular for software engineers.
- Like other central notions (e.g. information), the term "model" is hard to define.
- People using models should make sure that they do not confuse their models with the reality. And they should draw conclusions from their models only very carefully, taking into account the limitations of the models.
- Creating prescriptive models is easy, but greatly improved descriptive models are what we desperately need.

Acknowledgements. Three anonymous reviewers have contributed many useful objections and corrections. Michael Jackson in England has volunteered to read and comment my manuscript. Finally, Martin Glinz, the guest editor, helped to polish the final version. Many thanks to all of them for their time and effort!

References

1. Cameron, JR. (1986) An overview of JSD. IEEE Trans. on Softw. Eng. SE-12(2): 222–240
2. Kent, W. (2000) Data and Reality. North-Holland. Republished by 1stBooks Library (including an electronic version, see `http://www.1stbooks.com/`), April
3. Beck, K. (1999) Extreme Programming. Addison-Wesley, Reading, Mass.
4. Drappa, A. (2000) Quantitative Modellierung von Softwareprojekten. Dissertation, University of Stuttgart. Shaker-Verlag Aachen
5. Drappa, A., Ludewig, J. (2000) Simulation in Software Engineering Training. 22nd ICSE, Limerick, pp. 199–208
6. Georgescu, A. (2003) Web pages on SESAM, see `http://www.informatik.uni-stuttgart.de/ifi/se/research/sesam`
7. Jackson, M. (1995) The world and the machine. Proc. of the 17th ICSE, Seattle. ACM, pp. 283–292
8. Stachowiak, H. (1973) Allgemeine Modelltheorie. Springer-Verlag, Wien etc.