Software Design and Architectures
SE-2 / SE426 / CS446 / ECE426

Lecture 3 : Modeling Software

Software uniquely combines abstract, purely mathematical stuff with physical representation. There are numerous views or models of software possible, associated with the various design processes and methods.

Example : Some Models of a Desktop Application

Recalling the definition of *design* as *set of models* let's consider the modeling of some “real” software.

Consider a desktop application which maintains and displays calendars of events, appointments, and tasks. It displays data on the desktop, and accepts data from the user and from the network (for shared data).

[iCal Screen shot]

This is not a *very* complicated piece of software. In the spirit of our earlier discussion, let's look at some (incomplete!) models of this application.

Modeling the Data There are days and weeks and months, and there are Events which might be all-day events with a start-date and end-date; or appointment events with a date, start-time, and end-time. Of course this is not a *complete* model! This sort of model might be made very early (even during requirements modeling if interpreted as modeling the domain alone) to investigate "what if"'s and class-responsibility-collaborator analyses.

(This one is a “Class Diagram” in the UML. But it's important to realize that we don't insist on the *UML* in order to believe that we’re building *models*.)

[Calendar Data Model]

Modeling the Data Repository The Calendar data will eventually be stored somehow and somewhere. A user may have many Calendars, and will probably want to archive them, copy them, and do other similar things to them, so each Calendar might well be a file in the operating system. When the application starts it will want to know where the current Calendars are. Here are a couple of choices.

(This one is in a notation I invented for the purpose, just to make the point that the UML is not necessarily the only modeling language in the world. In fact it doesn't have a very convenient notation for discussing things like files and directories. Even though the Unified Process does

[Data Repository Models]

Modeling the Processing The user, the local system, and the network are all sources of events. We look at state transition of an Event which is being warned. Finite-state machines are of course not at all the only way to model processing.

[Event FSM Diagram]

Modeling the Source Repository The developer must understand how all this stuff is organized as source. We imagine a version without the network facilities and another with them.

(I have used a simple variation on the UML‘s packaging ("development") notation. In fact the UML's ‘package’ is a pretty general facility for grouping things.)

[Diagram]

Multiple Models
As the examples have shown, there are many different kinds of model which may be “built” when designing software. Sometimes we have a formal (or somewhat disciplined) design *process*, in which case different kinds of model are associated with design methods However, they can better be understood as *generic* and of value independently of the process which gives rise to them.

What we called a *model* in the general design setting is also (for software) sometimes called a *blueprint* (in the Rational tradition) or a *view* or a *representation*. Some authors are thinking of the software as concrete and the model/blueprint/view as derived from it (even though the software doesn't exist yet!) Some authors are deliberately (or carelessly) confusing the meaning (a particular mathematical structure) with the language or syntax (graphs, diagrams, etc.).

Best: *model* is what we already mean. *View* can be take as *class of models* as in Kruchten (below). *Representation* can be *model* when emphasising the syntactic aspect (e.g. this graph rather than that one).

To keep it clear: our construction will be largely ideal (having ideal / mathematical properties) but will have a physical aspect. We don't have it yet. Until we have it, we build *models* which are (recall):

> concrete enough to be communicable, evaluable, and related to the projected construction, while also being cheaper to produce and modify than the real thing

Software is pretty abstract, even in its physical. We tend to use graphs (aka box and arrow diagrams) to represent models of software concretely as in the example. The nodes of the graph are either software entities, or classes (sets, or in the UML, ‘classifiers’) of them, or states of them; and the edges are relationships between the entities, or sets; or they are transitions between states. Often the *layout* of the diagram, not just its connectivity, is meaningful. But unless we're using a formal diagramming notation (FSM's, SDL, UML, etc.) then we have to be careful, when *communicating*, that the intended meaning of our diagrams is clear. In pure modeling terms, we have to be careful that the relationship between properties of the model and properties of the projected construction, is clear.

Models and Attributes
The model of House Arrangement in a previous lecture we saw to ignore certain issues (such as the presence of electrical plugs) in the projected construction; and certain other attributes (such as the size, material) were not *relevant*, but accidental properties of the model.

A model may be considered as defining or specifying or correlating with certain *attributes* of the projected construction. As we consider different models we are considering different constructions (different because their attributes differ). That is the reason for modeling.

Always remember that the relationship between the model's properties and the projected construction's properties must be understood as part of the modelling process. Some model features are to be ignored; some are to be emulated. The ignorable ones are also those which shouldn't be part of any analysis.

[Budgen p 94: Resolving Attributes]

White Box and Black Box

A fundamental distinction in modeling anything, but especially software which is all concerned with structure and scope.

Some models are purposely intended to give an "external view", which may define the *appearance* of the projected construction. These may be called *black box* models because they aren't supposed to be seen inside, even though the model must have some discernable internal structure. The projected construction is only constrained by the external attributes of such a model.

Example: a picture of a building
Example: screens designed using a paint tool; using HTML authoring tool (understood as blackbox)
Example: a mathematical function to be implemented as a C procedure
Example: an API provided to define an interface, and implemented using a temporary (e.g. proprietary or ultimately inadequate) package.
Example: SDL spec (or indeed the earlier FSM) (understood as describing functional behaviour)

Some models are purposely intended to convey the "structure", even if their functionality as models is not complete. For these, what is seen inside constrains the projected construction. These may be called *white box* models, because they are supposed to be seen inside.

Example: electrical, plumbing diagrams for building
Example: class or interface diagram in UML (understood as whitebox)
Example: screens designed using HTML authoring tool (understood as whitebox)
Example: sequence diagram
Example: pseudocode
Example: SDL spec (understood as describing software structure)

Note that the same model entity might have a different interpretation depending on whether it's meant to be a white-box or black-box model. (Examples above suffice.) So it's essential to know which kind of model we're considering.

Software Entities and Relationships
There are many kinds of soft "entities" or unit which may be mentioned or invoked in assembling our models:

data structures, file structures, header files, XML sublanguages (Schema or DTD), CSS or HTML
external and internal events
thread, tasks, and processes
source files, make files, CVS repositories, packages and packaging frameworks
interfaces, containers
references, URLs
processors, peripherals, networks, protocols
states of the above
classifiers of the above

There are many relationships between them which may be mentioned:

sends/receives data
sends/receives control
sends/receives event
uses
depends on
inherits from
synchronizes with
runs in parallel with
executes

follows in sequence

Software has above all both a *static* aspect and a *dynamic* aspect. Both aspects are complex and constantly shifting. The static aspect shifts as the software evolves, under the control of the developers: the dynamic aspect shifts as the system interacts with its users or environment.

Design Views in the Design Process
The various views are more or less relevant at different times in the design process.

[Budgen diagram 5.6 p 95: Views and the Design Process]

Views: Classifying Models
Because there are so many potential models (views) involving so many attributes, methodologists are trying to classify them in order to give guidance on method.

Budgen says: data modeling, functional, behavioural, constructional.
Kruchten says: logical, process, physical, development, plus "scenarios to pull it together"
Rational says: logical, process, deployment, data, use-case, implementation
They overlap.

Here is an outline of the classifications:

Data modeling is concerned with the data objects user and the relationshiops between them. description of data structures: type, sequence, format. Critical to detailed design. The *logical* class: for Kruchten, the class or entity-relationship account of the functionality, using ideas from the problem domain (based on the Domain Model from the requirements). (It is what would be called the Logical model in database design.) It inherits the most material from the requirements specification: but now this logical view is in terms of actual software units. For Rational, includes the layers, subsystems, packages, etc., and seems to emphasise *software architecture*. Decision time: Kruchten emphasises early, Budgen emphasises late, for Rational (iterative development) this class of models is visited early (to establish core structure of the software) and late (to perform so-called Architectural Analysis.)

Budgen's *functional* view is “what the system does in terms of its tasks”. Actual “functional” specification. Kruchten's behavioural view includes causal issues (event/response). Finite-state machines. Sequence and statechart diagrams. Representation of time aspects. Can be black or white-box and the distinction is important. This view is not included in the Rational Unified Process because it is not “architectural” (too low level?)

The *process* view is concerned for Kruchten with *major tasks* and with *minor tasks*. Models in the major-tasks view address nonfunctional requirements such as performance and system availability, major concurrency, distribution, system integrity, fault-tolerance at the logical level. Models in the minor-task view address low-level concurrency (such as interrupt handling, UI event handling, display multithreading, and the like). Threads are assigned to tasks in logical view, and to processes in the physical view. Major tasks (processes) are viewed as “knowing each other” in other words having a static relationship defined by name; minor tasks (threads) are transient and belong to processes. The major-tasks partition is an aspect of *software architecture*, more to follow. A Broker- or Services-style architecture may make a system more dynamic and reconfigurable than this division suggests.

Kruchten's *physical* and Rational's *deployment* views model above all the assignment of tasks to processors and nodes of the network. This allows analysis of those nonfunctional requirements which relate to physical aspects, especially availability, reliability, performance, and scalability. These have ultimately physical aspects that can't be abstracted. Map elements from other views to physical components. Especially important in the presence of special hardware, networks, *etc.* It may be necessary to decide early, at least to make a prototype or proof-of-concept involving risky hardware

decisions: hard physical limits may intervene.

The development, constructional, or implementation view is the source respository and models of it. We may include the “build process” here as well (how the system is “built” from source). Data specification (programming language level: header files, tables, etc.) Threads and processes as language structures. Organization of actual software modules (packages, libraries, source files) in the environment, assignable to developers. Issues related to development management, reuse and commonailty, cost evaluation, project monitoring. Subsystem and module diagrams showing import and export relationshiops. These are later decisions, though they could be roughed in earlier.

White and Black According to View
An important aspect of a model is the degree to which it's understood as black- or white-box. This can be determined when you know the view which the model is meant to belong to. For example, a State diagram in the behavioural view is intended as black-box, in the sense that there is not intended to be any explicit representation of the ‘states‘ of the diagram within the projected construction; on the contrary. However, the same chart appearing in the constructional view may document an explicit finite state machine in the code.

Scenarios : Pulling it Together.
According to Kruchten, scenarios (use cases) are what pulls together the 4 other views: so for him the fifth view is essential in a special way. According to the Unified Process [*per* Craig Larman] the Use case view is just another view like the others. I prefer Kruchten's observation.

References

Garlan and Shaw pp 1 – 20.
P Kruchten, "The 4+1 View Model of Architecture", IEEE Software 12(6), Nov 1995, pp 42-50. (Available on-line within Waterloo, at http://ieeexplore.ieee.org/iel1/52/9910/00469759.pdf .)
Budgen Chh 5-7 (pp 89 - 168 in the second edition...).
Pressman Ch 12 (beginning of) and Ch 13 (beginning of).
Larman pp 500-503