Success Story

Redeveloping eVACS using xtUML and BridgePoint

Dr Malte Stien

For the 2001 ACT Assembly election the Australian company *Software Improvements* developed eVACS — the *Electronic Voting and Counting System* following a structured methodology and coding in C. A year later a portion of the system was re-engineered using *executable translatable UML (xtUML)* at the *Australian National University*. xtUML development occurred more than two-and-a-half times faster and all development artefacts were of high integrity.

For the 2001 Australian Capital Territory (ACT) Assembly election the Australian company Software Improvements Pty Ltd successfully developed a system that allowed voters to vote electronically. The so–called eVACS system consists of four sub–systems: for setting up the election, for conducting the actual voting procedure, for the data–entry of paper ballots, as well as a for counting both the entered paper votes and the electronic votes.

In 2002 a real–life example for teaching xtUML at the Australian National University was needed and found in the voting subsystem of eVACS. A single developer spent three months on redeveloping the voting subsystem using the xtUML–method and the BridgePoint Design Suite developed by ProjectTechnologies (www.projtech.com).

Voting in Australia

For all those not familiar with the ACT voting system this section contains a concise but incomplete introduction. In the ACT a voter can assign increasing preference numbers to election candidates to express an order of preference for one or more candidates. The first preference number is one and should be assigned to the candidate the voter prefers over all others. The next preference number is two and should be assigned to the next candidate in the order of preference of the voter. Preference numbers in increasing order can continue to be assigned by the voter to the maximum number of candidates.

A voter can vote at any polling place in any electorate irrespective of the electorate in which the voter is registered. As a consequence all polling places have to know about all the candidates and their political parties (if any) in all of the electorates.

At the 2001 ACT Assembly election, when a voter arrived at the polling place he or she was asked whether he or she wanted to vote electronically or via a paper ballot. To vote electronically, the voter was supplied with a bar code that ensured the voter could vote only once. To maintain anonymity the bar code was never linked to an actual voter. The voter then used the bar code to start the voting process and again at the end to confirm his or her vote. After that, the bar code was marked as used within the system and could never be used again.

The Redevelopment

A set of scenarios given as event–action–lists formed the basis for the redevelopment of the voting part of eVACS. As shown in Figure 1 the system was divided into three domains: a domain for the actual voting, another for guiding the voter through the process and a graphical user interface. While the first two were developed directly from xtUML–models the latter was handcoded in Java and interfaced with the voter guidance domain.

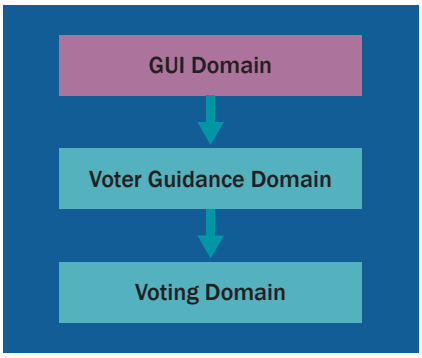

FIGURE 1: THE VARIOUS DOMAINS OF THE SYSTEM. Each arrow represents a dependency from one system to another system to which the arrow is pointing. The green systems have been realised using xtUML while the purple system has been implemented using traditional handwritten Java.

The Voting Domain

The voting domain consists of three subsystems that separate areas of concern within a domain: one subsystem contains the actual domain classes that represent all the concepts of voting such as groups, candidates, electorates, polling places, votes and preferences. Another one contains specification for setting up the system during start–up and the third subsystem contains tests for verifying the model.

Figure 2 shows a small portion of the actual domain classes within the voting domain. Most of the classes within the domain such as candidate and preference are passive, i.e. they

do not have a complex lifecycle represented as a state machine but rather exist for the purpose of storing information that needs to be related to other objects or for providing synchronous operations. Within the entire domain only the vote class features a complex lifecycle as can be seen in Figure 3 which shows the entire statemachine of the vote. After a temporary vote has been created it might be edited by the voter. While it is in the Editing Vote state preferences can be added to or deleted from the temporary vote. The voter might even initiate the deletion of all preferences assigned to the temporary vote and so begin voting again with a blank ballot. Within the Delete All Preferences state the vote does exactly that: it dissociates all preferences from itself. Afterwards, the temporary vote generates the Advance event to itself to advance back to the Editing Vote state. If the voter made all the choices for his or her vote he or she might want to commit the vote by generating a corresponding event. Now the vote is in its permanent quiescent state. The instance of the vote still exists but its lifecycle has reached its final state. The vote can never be edited again. Clearly, this behaviour reflects the common policy for votes. One of the requirements for the system was that — as it is on paper — making an informal vote shall also be possible. Thus, the voter can commit the vote directly after it has been created without editing it.

Note, that the domain contains no information at all about how the voter interacts with the system. The domain does not even contain information about the fact that the voter selects a candidate before voting for him or her. The sole purpose of the voting domain is to capture all business policies to do with voting. Selecting candidates does strictly not have anything to do with voting itself but with guiding a voter through a possible voting process.

Some of the classes, such as preference, feature a so called assigner state machine. That is a state machine that is linked to the class rather than an instance of a class (similar to a *static* method in C++ or Java). An example is given in Figure 4. Within the Adding Preference state, preferences get added to the currently edited temporary vote, i.e. an instance of preference gets created and associated with the vote. Within the Deleting Preference state the last assigned preference of the currently edited vote gets unrelated from the vote and deleted. In both states, both of the events that trigger one or the other action may be processed. The event AddPreference carries the group index letter which unambiguously identifies a group as well as the position of the candidate on the ballot paper. The event DeleteLastPreference does not carry any parameter as the last preference added to the current vote can be retrieved from the system.

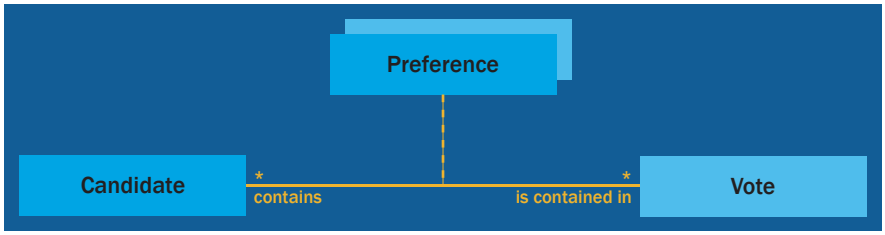

FIGURE 2: AN EXCERPT FROM THE CLASS DIAGRAM OF THE VOTING DOMAIN. The classes *Candidate* and *Preference* are passive while *Vote* is an active class, i.e. it features a state machine. However, the class *Preference* has an Assigner that manages the creation and deletion of temporarily assigned *Preferences*.

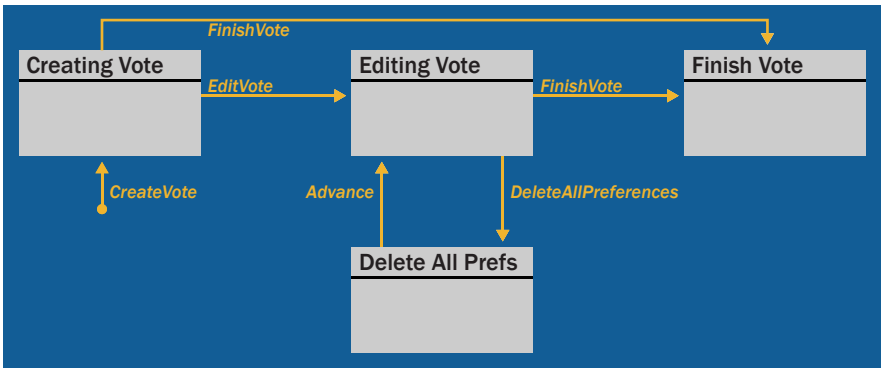

FIGURE 3: THE LIFECYCLE OF AN INSTANCE OF THE CLASS VOTE. The yellow arrows represent transitions between states.

The Voter Guidance Domain

Like the voting domain this domain also consists of three subsystems: The actual voter guidance classes, a subsystem for setting up the system during initialisation, and another subsystem for testing. The main part of this domain is a class that contains a comparatively large state machine representing the sequence of steps in which a voter might vote. This state machine represents concepts such as selecting a candidate, switching between groups and assigning the next preference number to the currently selected candidate. The domain has no concept of how the next preference number is determined as this is a matter of voting as opposed to guiding the voter through the process. Hence, it requires the preference number from the voting domain.

Note, that even this domain still contains no concerns about the means a voter might use to interact with the system. The voter might use a mouse, a keyboard or a voice recognition system. This is left to the GUI domain. Objects in the voter guidance domain merely react on events that are emitted from the GUI. Those events represent for example "The voter demands to assign the next preference number to the currently selected candidate", "The voter demands to undo the last step" or "The voter demands to finish the vote". In some cases objects in the user interface might send events to objects in the voter guidance domain which cannot be processed while the system is in a certain state. According to the rules of xtUML some events that cannot happen are ignored. For example, the voter should not be able to delete the last preference after finishing his/her vote.

The developer used the BridgePoint ModelBuilder to construct the analysis model in a graphical environment. Classes have been drawn and related to each other by using a graphical user interface. Action Language, attribute

names and types have been entered textually.

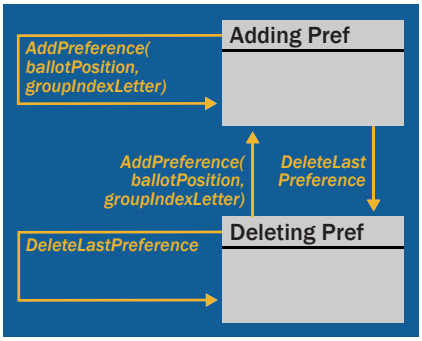

FIGURE 4: STATE MACHINE OF THE PREFERENCE CLASS' ASSIGNER. At any point in time a *Preference* can be added or the last added *Preference can be deleted.*

Verification of the Model

While the process of modelling the two domains was still ongoing the developer used the BridgePoint ModelVerifier from time to time to verify the correctness of the model at particular points in time. For this purpose each of the two domains contain a test subsystem which consists of several test voters each of which feature different behaviour. As a kind of regression test vote tests were executed on the model to verify its behaviour. Similar to a syntax checker of a traditional programming language the ModelVerifier detects inconsistencies as well as some other errors within the model. Unlike a syntax checker which works statically on the code the ModelVerifier is a dynamic verification tool and detects errors during execution of the model in a virtual machine environment. The developer was able to stop the execution at any point in time to inspect or modify member variables of any of the class instances, the event queue, or change any of the relationships between class instances. The ModelVerifier can execute a model as fast as possible or step by step.

To execute the model, no code had to be generated. Even if the target architecture is not determined the model can be fully executed and verified.

Compilation of the Model

The ModelCompiler MC–2010 (by ProjectTechnologies) transformed the object–oriented analysis models for the voting and the voter guidance domains into executable machine code. This has been performed fully automatically in a two stage process: First the code got transformed into compilable C++ code. The ModelCompiler provided some additional runtime libraries and fed both into a standard C++ compiler. The resulting code is an executable that runs under the SOLARIS operating system and communicates with the outside world via UNIX message queues. Both domains are contained within the same executable.

The GUI Domain

The graphical user interface was handcoded in Java using the Swing library. An important point is that due to the separation of domains the GUI had no concept of voting, political groups or candidates.

In Figure 5 the overall sequence of voting with the system is shown. On the Main Voting Screen the GUI simply displays columns of political groups each containing a number of candidates. The voter can use arrow keys on a keypad to navigate between candidates and use a special defined key to trigger the selection of particular candidates. For every key stroke, the GUI merely generates an event informing the voter guidance domain that the voter pressed a particular key. The GUI then may or may not receive an event from the voter guidance domain commanding it to perform a certain operation such as deselect or select a particular candidate, move the cursor to a particular candidate or simply display a preference number.

The Java code utilized a small shared library that has been written in C. This module connected to the Java code using Java's Native Interface technology (JNI) as well as to the executable containing the voter guidance and the voting domain using UNIX message queues. The shared module was necessary because the developer wanted to make use of a small code module that the ModelCompiler provided to enable an external application to talk to the generated executable. See Figure 6 for an overview of the architecture.

Conclusion

The application of xtUML led to a significantly reduced development time. Even though the developer had to learn the executable translatable UML method, he implemented the voting system — which represents about 40% – 50% of the entire system — in 65 days. Given that the original project team took 365 man–days to build the entire application xtUML accelerated the realisation of the voting system by a factor of at least 2½. In both, the traditional and the xtUML approach the given numbers exclude the requirements gathering phase.

Once the code was generated, it was basically defect free. All the

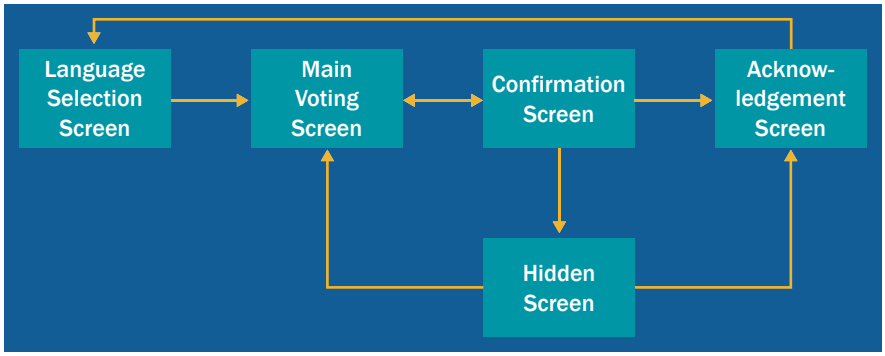

FIGURE 5: THE SEQUENCE OF THE VOTING PROCESS. After choosing the language for the system messages the voter can assign preferences to candidates on the *Main Voting Screen*. Afterwards, all preferences get summarized on the *Confirmation Screen*. From there the voter can either swipe a barcode to confirm the vote or return to the *Main Voting Screen* to keep editing the vote. If the voter encounters any problems he or she may switch to a *Hidden Screen* to call an official for assistance. From there, the voter has the same possibilities as from the *Confirmation Screen*. From the *Acknowledgement Screen* the system returns automatically to the *Language Selection Screen* to welcome a new voter.

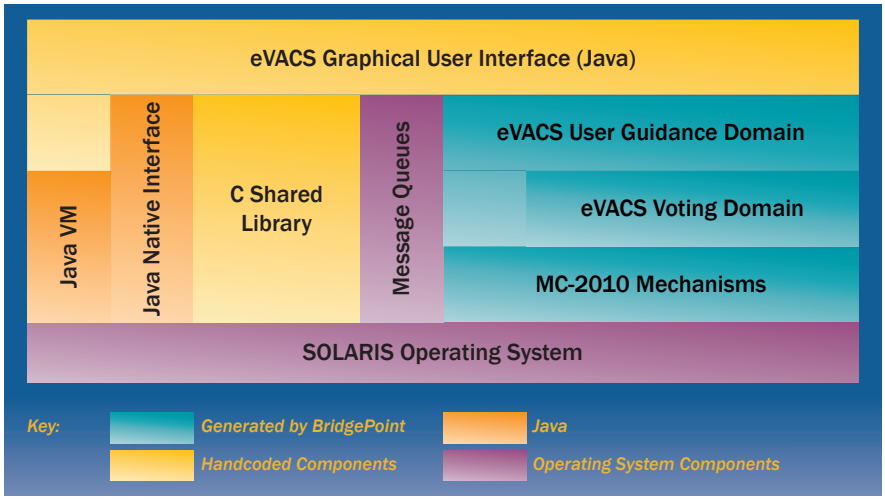

FIGURE 6: THE OVERALL ARCHITECTURE OF THE VOTING SYSTEM. See text for details.

conceptual defects had been found during verification of the model using the ModelVerifier. Other defects, that normally occur due to the implementation language, such as null pointer exceptions or uninitialized variables, did not occur. The architecture which is part of the ModelCompiler takes care of those issues.

One of the major aims of xtUML is to separate the analysis of a real–world problem from the architectural aspects of the implementation. This worked perfectly in this case study. While the developer could fully concentrate on the analysis of the voting and voter guidance problem domains the ModelCompiler generated the entire code for each domain. The code then interfaced nicely with the graphical user interface which had been developed separately in Java.

ModelCompilers are available for a limited number of target architectures. They might work slightly differently and are generally open to changes that might be necessary to adapt the ModelCompiler to the requirements of a particular target system. Even if such tailoring has to be done the developers get the full benefit of the xtUML methodology.

Literature

The following books provide further information on xtUML (formerly known as the Shlaer/Mellor method):

- Mellor, Stephen J. and Marc J. Balcer, Executable UML – A Foundation for Model–Driven Architecture, Addison–Wesley, 2002
- Starr, Leon, Executable UML, The Models Are the Code – A Case Study, The Elevator Project, Model Integration LLC, 2001
- Starr, Leon, Executable UML, How to Build Class Models, Englewood Cliffs, NJ, Prentice/Hall, 2001
- Starr, Leon, How to Build Shlaer–Mellor Object Models, Prentice Hall, 1996
- Shlaer, Sally and Stephen J. Mellor, Object Lifecycles – Modelling the World in States, Prentice Hall PTR, Englewood Cliffs, N.J., 1992
- Shlaer, Sally and Stephen J. Mellor, Object/Oriented Systems Analysis – Modelling the World in Data, Prentice Hall, Englewood Cliffs, N.J., 1988

Dr Malte Stien graduated in 1998 with a degree in Technical Computer Science in Berlin/Germany. He worked formally in the field of surgical robotics, the subject of his PhD thesis which included simulation and monitoring of surgical robotics tasks. After coming to Australia in 2002 he worked for the Australian National University on the project described in this article. Dr Stien works for Software Improvements as a Senior Software Engineer where he provides consultancy services to BridgePoint clients. Feel free to contact him via stien@ieee.org.

Software Improvements aims to provide quality products that ensure their use by customers results in ethical and defensible outcomes.

Established in 1992, *Software Improvements* specialises in improving processes of acquisition, development, operation and maintenance typically used by developers for critical software systems thereby reducing the risks and overheads for purchasers of software.

For more information on eVACS, the BridgePoint Design Suite or xtUML training contact

Software Improvements
PO Box 1928
Canberra ACT 2601
Australia
T: +61 2 6273 2055
F: +61 2 6273 2082
www.softimp.com.au
info@softimp.com.au

Software Improvements