ELSEVIER

The Journal of Systems and Software 61 (2002) 105–119

The Journal of Systems and Software

www.elsevier.com/locate/jss

Design erosion: problems and causes

Jilles van Gurp *, Jan Bosch

Department of Mathematics and Computing Science, University of Groningen, P.O. Box 800, 9700 AV Groningen, The Netherlands

Received 4 June 2001; received in revised form 21 September 2001; accepted 20 November 2001

Abstract

Design erosion is a common problem in software engineering. We have found that invariably, no matter how ambitious the intentions of the designers were, software designs tend to erode over time to the point that redesigning from scratch becomes a viable alternative compared to prolonging the life of the existing design. In this paper, we illustrate how design erosion works by presenting the evolution of the design of a small software system. In our analysis of this example, we show how design decisions accumulate and become invalid because of new requirements. Also it is argued that even an optimal strategy for designing the system (i.e. no compromises with respect to e.g. cost are made) does not lead to an optimal design because of unforeseen requirement changes that invalidate design decisions that were once optimal.

1. Introduction

With the ever-increasing size and complexity of software, the weaknesses of existing software development methods and tools are beginning to show. This is particularly true when it comes to maintaining the software. As early as 1968 the software crisis was identified during a NATO workshop (Naur and Randell, 1969). Since that moment, many approaches have been suggested to solving the software crisis, many of which are still applied today. In this paper, we intend to illustrate that despite 30 years of research and despite the many suggested approaches, it is still inevitable that a software system eventually erodes under pressure of the ever-changing requirements.

Recent examples of approaches are the architecture development method discussed in Bosch (2000), the software development method Extreme Programming (Beck, 1999) and many others. However, we have reasons to belief that such approaches still do not fully address the issues identified in 1968. The example we present in this paper serves both as an illustration of design erosion and related problems and as a starting point for future research. Furthermore, we present two strategies for incorporating change requests: the optimal architecture strategy and the minimal effort strategy.

1.1. Industrial examples

Design erosion is quite common and the diagnosis of its occurrence is often used as a motivation for redeveloping systems from scratch. In most cases, such redevelopment requires a massive effort. An example of a project where this happened is the Mozilla web browser. Three years ago, Netscape was experiencing fierce competition from Microsoft's Internet Explorer. They decided to release their own browser as open source and started working on transforming it into the next generation browser. After half a year of development, the developers of the open source Netscape concluded that the original Netscape source was eroded beyond repair. They took a major decision and started from scratch. Now, more than two years later the Mozilla project is still working on this browser. An enormous amount of code has been released and some of it has been retired yet again (despite it being written from scratch). An example of this is the caching component, which was recently replaced by a completely new version because of less than optimal design decisions in the original version. Apparently during the two years of redevelopment, requirements had changed sufficiently to retire a part of the system before the system was even finished.

* Corresponding author.

E-mail addresses: jilles@cs.rug.nl http://segroup.cs.rug.nl (J. van Gurp), jan.bosch@cs.rug.nl http://segroup.cs.rug.nl (J. Bosch).

0164-1212/02/$ - see front matter
PII: S0164-1212(01)00152-2

A second example of design erosion we have encountered (Bosch, 1999; Svahnberg and Bosch, 1999) is the Axis case. Axis AB is a Swedish company that produces network devices that replace PC's as a means to offer network connectivity for common PC peripherals like printers, scanners, CD-ROMs, ZIP drives, etc. In the early days of this company, this company only had a printer server, however, support for other devices was added over time. At some point the developers realized that in order to support new types of devices, a radical restructuring of their software was needed. Rather than patching up the existing software, it was decided to build a new architecture. After two years of development (while simultaneously maintaining the old software), they were ready to release products based on the new software. When we recently visited Axis, we found out that this new architecture (after a few years of successful use) was being replaced by a third generation of software (they were migrating from their proprietary OS to an embedded Linux version).

A third example is the Linux kernel. Like Mozilla, this product is developed as an open source project. One of the reasons it took nearly two years to develop kernel 2.4 (which was released recently) after the previous stable release (version 2.2, odd version numbers like 2.3 are considered to be development versions) is that much of the old 2.2 code needed massive restructuring in order to incorporate the new requirements. By redesigning large parts of the old kernel, the performance was enhanced and new requirements could be met. A similar effort can be expected for the next release (i.e. version 2.6).

In these three examples, the redevelopment of the software can be considered a success. However, considering the effort needed to do so, it can easily be imagined that some companies are less fortunate in identifying the signs of design erosion early enough to be able to take such action. Redeveloping software (also referred to as the revolutionary approach), is a very expensive and lengthy procedure and failing to see it is necessary can be fatal to a software producing company.

A second issue that we have observed is that in all three cases, the redevelopment of the software was only partly successful. Mozilla has already seen some of its components rewritten, Axis is already working on its third generation of software and the Linux development can be characterized as a continuous effort to perfect the system, often resulting in large parts being replaced by new code.

1.2. Problems

Based on the industrial cases that we have studied (e.g. Bengtsson and Bosch, 1998; Bosch et al., 1999), and the above examples, we have identified that design erosion is caused by a number of problems associated with the way software is commonly developed.

- *Traceability of design decisions.* The notations commonly used to create software lack the expressiveness needed to express concepts used during design. Consequently, design decisions are difficult to track and reconstruct from the system.
- *Increasing maintenance cost.* During evolution maintenance, tasks become increasingly effort consuming due to the fact that the complexity of the system keeps growing. This may cause developers to take sub-optimal design decisions either because they do not understand the architecture or because a more optimal decision would be too effort demanding.
- *Accumulation of design decisions.* Design decisions accumulate and interact in such a way that whenever a decision needs to be revised, other design decisions may need to be reconsidered as well. A consequence of this problem is that if circumstances change, developers may have to work with a system that is no longer optimal for the requirements and that cannot be fixed cheaply.
- *Iterative methods.* The aim of the design phase is to create a design that can accommodate expected future change requests. This conflicts with the iterative nature of many development methods (extreme programming, rapid prototyping, etc.) since these methodologies typically incorporate new requirements that may have an architectural impact, during development whereas a proper design requires knowledge about these requirements in advance.

1.3. Optimal vs. minimal approach to software development

Assuming an iterative development method, we can distinguish two stereotypical strategies for incorporating change requests into a software system:

- *Minimal effort strategy.* Incorporate the change in the next iteration of the development while preserving as much of the old system as possible. The advantage of this approach is the relatively low cost of each iteration. However, the accumulation of design decisions in each subsequent iteration limits what is possible at a reasonable cost in future iterations.
- *Optimal design strategy.* Make all the necessary changes to the software artefacts to get an optimal system for the new set of requirements. In principle, no compromises between cost and quality are to be made. The advantage of this approach is that the changed system is optimal for the requirements because any conflicts with decisions in the previous version are resolved. This means that future changes can be incorporated at a relatively low cost. However, re-

designing a system can take a lot of time and generally takes a lot of effort.

Both strategies are infeasible in general. The minimal strategy, because that causes problems for future changes. The optimal strategy, because the cost is too high. However, we tend to look upon these strategies as two extremes in a spectrum of approaches.

1.4. Related work

In Perry and Wolf (1992), a distinction is made between architecture erosion and architectural drift. *Architectural erosion*, according to Perry and Wolf, is the result of 'violations of the architecture'. *Architectural drift*, on the other hand is the result of 'insensitivity to the architecture' (the architecturally implied rules are not clear to the software engineers who work with it). Parnas (1994), in his paper on software aging, observes similar phenomena. Although he does not explicitly talk about erosion, he does talk about aging of software as the result of bad design decisions, which in turn are the result of poorly, understood systems. In other words: erosion is caused by architectural drift. As a solution to the problem Parnas suggests that software engineers should design for change, should pay more attention to documentation and design review processes. He also claims that no coding should start before a proper design has been delivered.

In Jaktman et al. (1999), a set of characteristics of architecture erosion is presented. Some of these characteristics are also identified in our own case study. In their case study, Jaktman et al. aimed to gain knowledge about how architecture quality can be assessed. Assessing architecture erosion is an integral part of this assessment.

To avoid taking bad design decisions, developers can consult a growing collection of patterns (e.g. Gamma et al., 1995; Buschmann et al., 1996). An approach to countering design erosion is refactoring (Fowler et al., 1999). Refactoring is a process where existing source code is changed to improve the design. Fowler et al. present a set of refactoring techniques that can be applied to a working program. Using these techniques violations of the design can be resolved. Unfortunately, some of the refactoring techniques can be labour intensive, even with proper tool support (e.g. Roberts et al., 1997).

Yet, another approach is to pursue separation of concerns. By separating concerns, the effect of changes can be isolated. E.g. by separating the concern synchronisation from the rest of the system, changes in the synchronisation code will not affect the rest of the system. Examples of approaches that aim to improve separation of concerns are Aspect Oriented Programming (Kiczalez et al., 1997), Subject Oriented Programming (Harrison and Ossher, 1993) and Multi Dimensional Separation of Concerns (Tarr et al., 1999).

1.5. Contributions and remainder of the paper

In many of the suggested approaches towards (e.g. Parnas' suggestions) solving the software crisis, it is assumed that if engineers work harder and/or more efficiently and/or use better tools, the problems will disappear. We disagree with this assumption and we demonstrate in this paper that design erosion is inevitable because of the way software is developed. Good methods only contribute by delaying the moment that a system needs to be retired. They do not address the fundamental problems that cause design erosion. Rather than fight the symptoms of design erosion we should start to address the causes.

In the remainder of this paper, we will discuss an example system (Sections 2 and 3). The reason for using a small example rather than an industrial case is that often companies are not in a position that enables them to follow an optimal strategy (which is what we do in the example). In addition, industrial cases may simply be too complex for our purposes. The advantage of the example we use in this paper is that we are in control of its development and that it is small enough to discuss in full detail. In Section 4 we present an analysis of our experiences with the example and we revisit the problems identified in this section. Finally, we conclude the paper in Section 5.

2. The ATM simulator

The example we present in this paper can be characterized as following a near optimal strategy for evolving a system (we have made some compromises). In our analysis, we show how the design decisions affect the system. In Section 3 we also reflect on what would have happened if we followed the minimal strategy for evolving the system. Economic concerns would probably have prohibited following the optimal strategy if our system had been larger, so it is worthwhile to examine both strategies.

The example we use in this paper is a simulator of a bank machine. The functionality of an automated teller machine (ATM) can be nicely expressed as a finite state machine (FSM), see Fig. 1. The start state of the FSM is `wait`. When in the `wait` state the FSM waits for a bankcard to be inserted. When a card is inserted, it is verified whether it is a valid card or not. If it is a valid card, the PIN code is asked and checked (maximum of three times, after three attempts the card is destroyed), after a valid PIN code has been entered, an amount of money needs to be given to the ATM. After a valid amount has been entered, the card is ejected and money

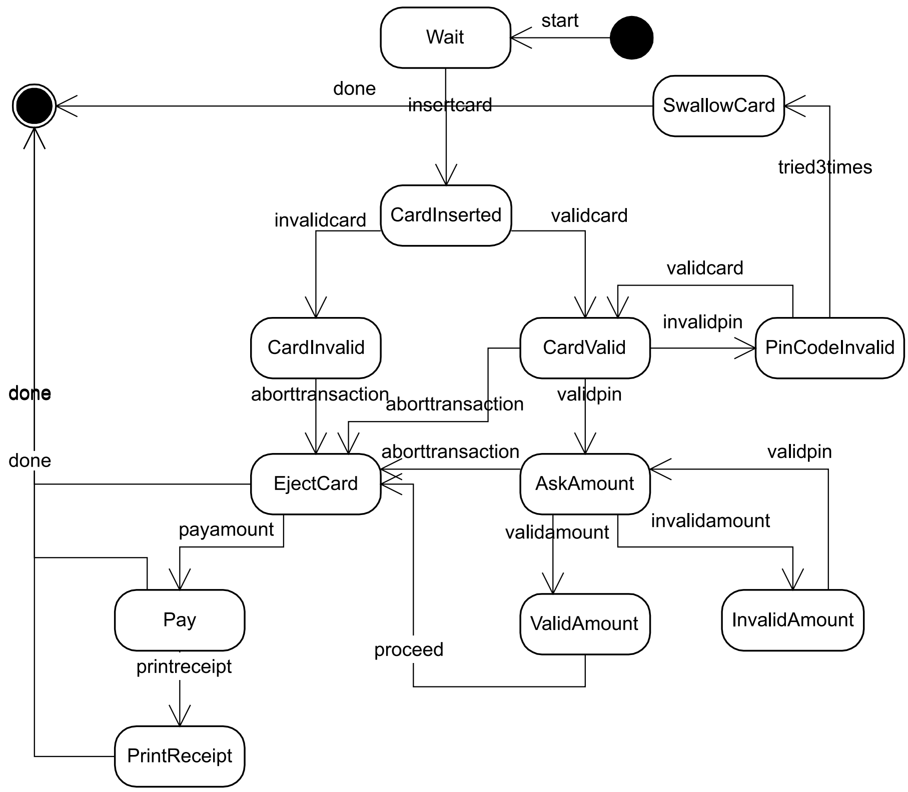

Fig. 1. ATM FSM.

is given to the client. Optionally, a receipt is printed. We have implemented several versions of the ATM Simulator. For each version, we introduced new requirements that forced us to redesign the system.

2.1. Version 1: the state pattern

2.1.1. Requirements

In the first version of the ATM Simulator, we focused on getting the system to work as specified in the FSM (Fig. 1). Our initial requirements were:

- *Core functionality*. Provide a simple implementation of the ATM Simulator, based on the specification in the FSM.
- *User interface*. Provide a primitive user interface to allow users to interact with the simulator.

2.1.2. Initial design

The first version of the simulator is based on the State pattern, which is described in Gamma et al. (1995). In Fig. 2, a diagram illustrates the structure of a State pattern application in our simulator. In the State pattern, a state machine's states are implemented as subclasses of a State class. A Context class is responsible for maintaining a reference to the current state (i.e. an instance of a subclass of State). State transitions are implemented as methods in the State subclasses.

Consequently, the design of the first version of our simulator contains an ATMContext class responsible for dispatching the events from the ATM FSM to the right ATMState instance (there are 12 subclasses, one for each state). In addition, the ATMContext class also stores any variables used by the ATMState subclasses. The reason for doing so is that these variables need to be shared between the various state classes (i.e. they are part of the context). A consequence is that a reference to the context needs to be available when events are dispatched. Because of this, the ATMState class has a property `context` that stores a reference to the ATMContext. Whenever a subclass needs to access one of the shared variables, it can access them through this property.

2.1.3. Issues

A few issues may cause maintainability problems:

- The ATMContext contains many methods that do nothing else but forward the call to the current state.
- ATMState subclasses inherit empty method bodies for all events in the FSM. Consequently, each state can process any event, even though the FSM specifies only a few per state.
- The ATMContext does not check whether a particular event is supported by the current state. It is the programmer's responsibility to check that events are processed in the right order.

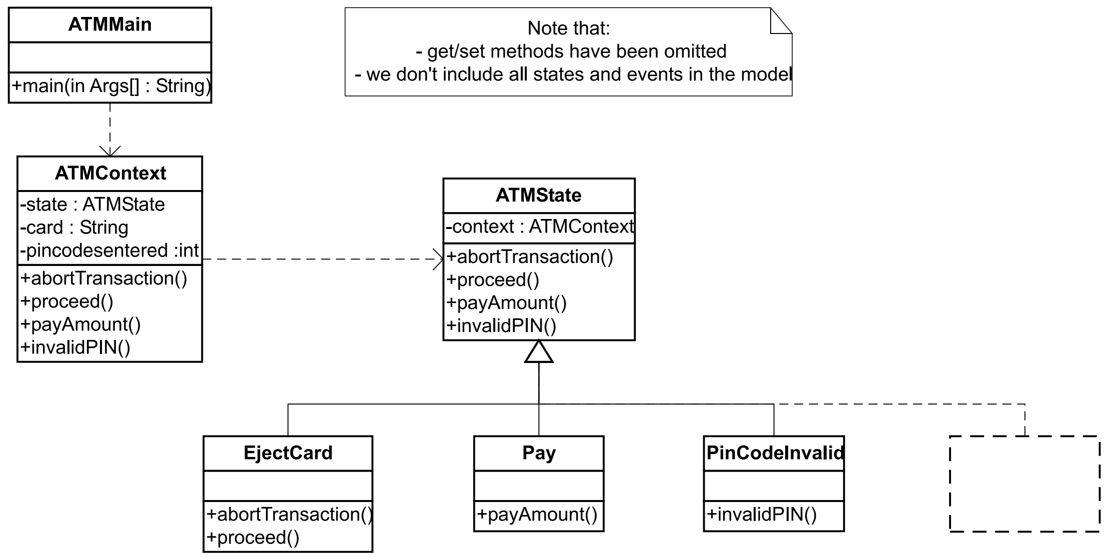

Fig. 2. Version 1: The State pattern in the ATMSimulator.

2.2. Version 2: the flyweight pattern

2.2.1. New requirements

In version 2 of the ATMSimulator, we focused on reducing the overhead of creating objects. Each time an ATMSimulator object is created, an object is created for each of the states. Recreating these objects is a time consuming and essentially redundant action. This is especially true since the state classes in version 1 do not store any data. The changes in this version address the following quality requirements:

- *Memory usage*. The aim of the changes is to instantiate the state classes only once.
- *Performance*. By reusing the state class instances, initialisation time of the simulator is reduced for subsequent uses after the first initialisation.

2.2.2. Changes

To allow for more than one instance of a FSM efficiently, the State pattern can be combined with the Flyweight pattern. This is also described in Gamma et al. (1995). In Fig. 3, the changed version of the model in Fig. 2 is displayed. The Flyweight pattern makes it possible to reuse instances of a class throughout a program. Consequently, only one instance is needed. Because the instances are shared, any data stored in the instance is also shared. Gamma et al. distinguish between intrinsic and extrinsic object state (not to be confused with a finite state machine's states). Intrinsic object state can be shared whereas extrinsic object state has to be provided to the Flyweight instance each time it is used. Luckily, the State objects in the ATMSimulator do not have any data that cannot be shared between multiple instances of the simulator except for the context property, which helps the methods in the state find, the context object containing variables that are needed in state transitions. So, little rearchitecting is needed in the state classes.

We removed the context property from the ATMState and inserted a context parameter in each event method. In addition, we made the shared instance variables in ATMContext static. These shared variables contain references to the state objects. Making these variables static causes them to be instantiated only once. This greatly reduces the number of objects in the system (if more than one instance of FSMContext is used). Without this change, each instance of FSMContext would create 12 state objects.

2.2.3. Problems and issues

A consequence of the flyweight pattern is that the state classes cannot hold any data (except for global data) since the instances are shared between the finite state machines. In our case, most of the data already resided in the FSMContext class, so that was no problem. A more serious issue was that version 1 used stdin and stdout for communication with the user. In case of multiple instances, these resources also have to be shared. We delayed solving this issue to version 3.

2.3. Version 3: multiple instances + new GUI

See Fig. 4.

2.3.1. New requirements

In this version, we evolved version 2 in such a way that multiple simulators can be run in parallel. Running multiple ATMSimulators may be useful if we move to a client–server architecture where multiple clients connect

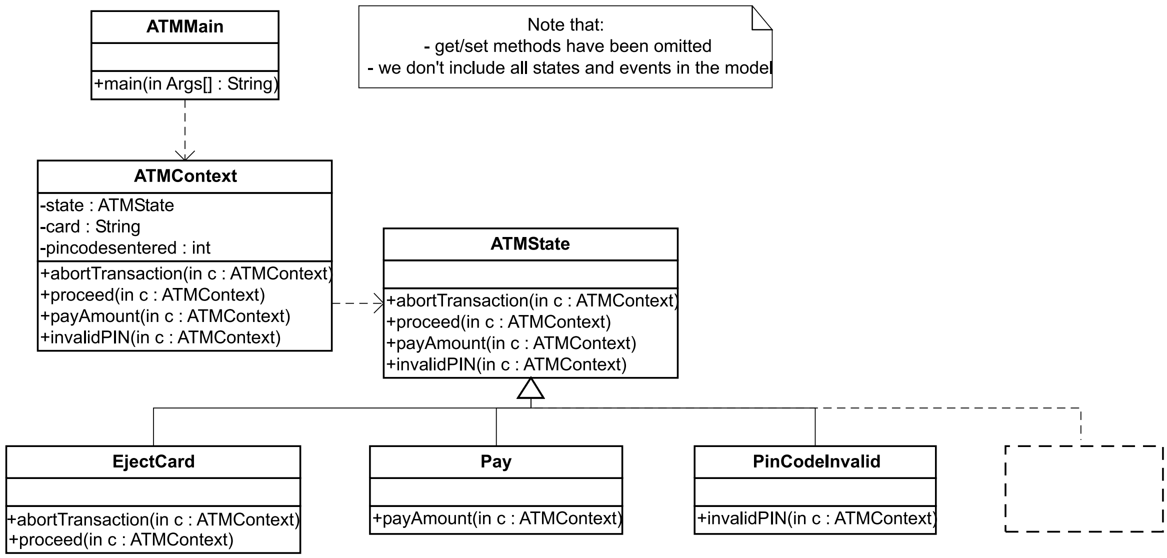

Fig. 3. Version 2: The Flyweight pattern in the ATMSimulator.

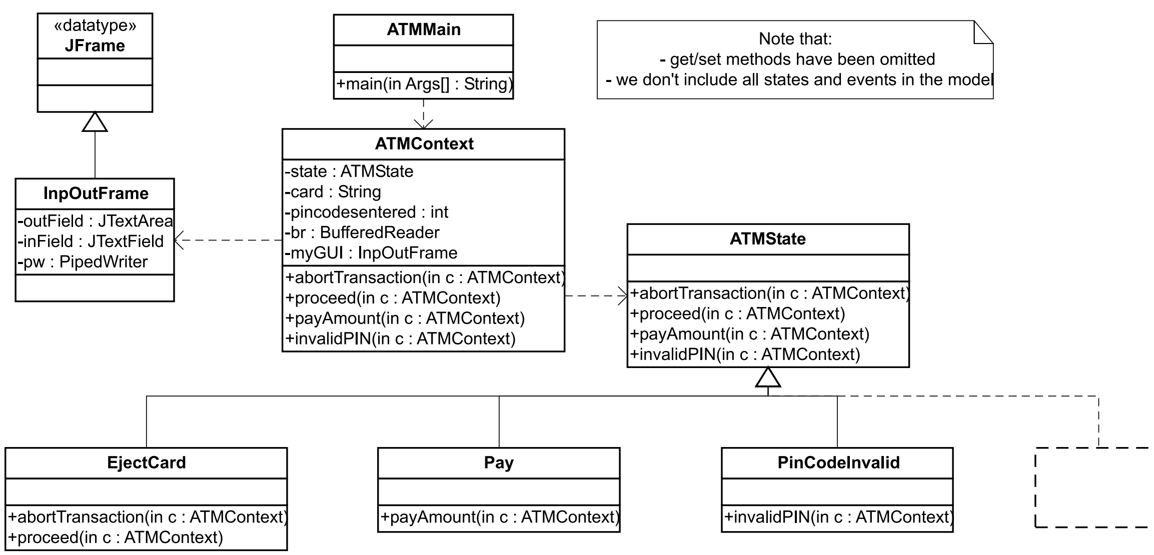

Fig. 4. Version 3: Multi user version.

to a server running the simulators. The previous version already made it efficient to create multiple simulators. However, the way user interaction was dealt with in that version made it hard to use more than one instance. This issue is dealt with in this version. The following functional requirements are addressed in this version:

- *User interface*. The user interface in the first two versions uses the command line for user input. However, when more than one simulator is used, a command line interface is no longer sufficient.
- *Parallelism*. By making each simulator a thread, it is possible to run them in parallel.

2.3.2. Changes

To address the user interface issues in version 2, we replaced the command line interface with a GUI. The GUI consists of multiple windows, each containing a text area for the output and a text field for the input. Each window is associated with an FSMContext instance. The GUI is connected to the FSM using a pipes and filters architecture. The reason we designed the system this way is that it allows us to preserve most of the code in the previous versions. Whenever a user enters text into the text field, this string is inserted into a pipe. The ATMSimulator can read from the pipe as if it were a regular IOStream (i.e. using readLine). Since it was previously reading from the stdin stream in a similar fashion, few changes were needed in the system.

In addition, we implemented the `java.lang.Runnable` interface in ATMContext. This interface makes it possible to create a thread from an object. Implementing the runnable interface has as a consequence that a `run()` method needs to be added. In the new version of ATMContext, this method only feeds new start events to the simulator. This causes the simulated ATM to run continuously.

2.3.3. Problems and issues

The system bypasses the model view controller architecture that is commonly used in Java applications. This may become a problem when we want to integrate our system with other systems.

2.4. Version 4: delegation-based approach

2.4.1. New requirements

In version 1 we already observed that there were some maintenance problems with the State pattern. In this version we have added a requirement for run-time configuration. This feature can be useful for dynamically reconfiguring of the system. In our ATMSimulator, for instance, it might be necessary to disable the receipt feature when the machine runs out of paper. Such a dynamic change can be modelled by rewiring a few arrows in the FSM describing the simulator. Making such changes in the FSM at run-time forces us to abandon the State pattern since this pattern relies on an implementation-time technique, inheritance, for adding states and transitions. The following requirements were addressed in this version:

- *Configurability*. Allow for run-time configuration, we want to be able to add new states and transitions at run-time.
- *Separation of concern*. In the previous versions, we noticed that the details of the ATMSimulator get mixed with the typical behaviour of finite state machines. Somehow, it should be possible to keep the two separated.

2.4.2. Changes

We refactored the system to use delegation instead of inheritance (see Fig. 5). This design decision is based on our earlier work presented in (Van Gurp and Bosch, 1999). Unfortunately, this change turned out to be quite radical. Rather than sub-classing ATMState, the class is instantiated when a new state is needed. Also, state transitions now have a first class representation (i.e. the FSMTransition class). Each state has a list of transition event pairs and a dispatch method that looks up the correct transitions for incoming events. Transitions, in turn delegate their behaviour to FSMAction classes. The latter is an incarnation of the Command pattern (Gamma et al., 1995). The intention of this pattern is to delegate behaviour to a subclass of FSMAction that implements specific behaviour. This way, the behaviour is separated from the control flow.

Furthermore, it was trivial to model state entry and exit events, which are commonly used in FSM specifications, so we added FSMActions that are executed when these events occur. We used this design solution to re- implement the ATMSimulator. Much of the code in the original FSMState subclasses could be copied into the FSMAction subclasses.

The changes are outlined in the diagram in Fig. 5. A small code example of how the framework is presented below. The AbstractFSMAction used in the example is a

```
public class ATMSimulator extends FSMContext {
static FSMState ejectcard = new FSMState("ejectcard");
static FSMState pay = new FSMState("pay");
static FSMState pincodeinvalid = new FSMState("pincodeinvalid");
static FSMState cardvalid = new FSMState("cardvalid");
...// more state definitions
static { // static -> it's executed only once
  pincodeinvalid.setInitAction(
    new AbstractFSMAction() { // Inner class definition
      public void execute(FSMContext fsmc) {
        ...// desired behavior
      }
     });
pincodeinvalid.addTransition(cardvalid, new DummyAction(), "validcard");
  ...// more transition and action definitons
}
...//rest of the class
}
```

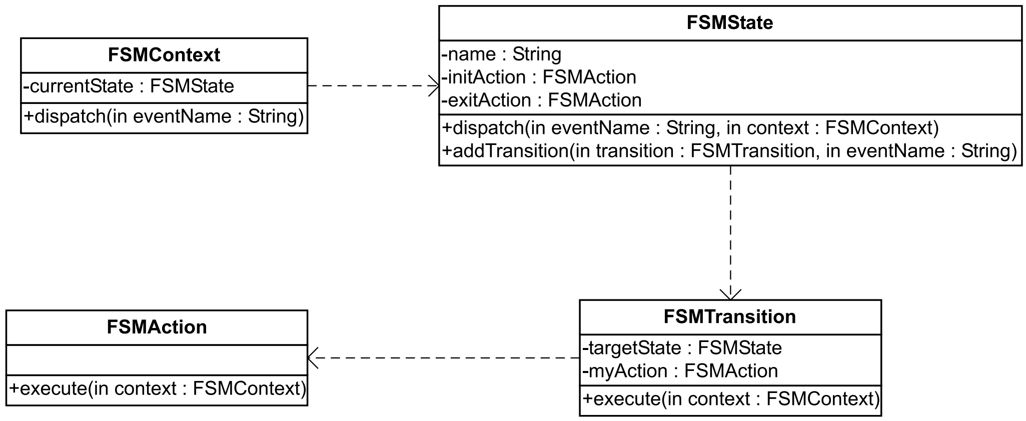

Fig. 5. Version 4: A delegation-based approach.

class that implements the FSMAction interface. This makes it easier to create inner classes for FSMActions. In the example, the three states we used before are created as FSMState instances. After that, we add an initAction to one of them and use this state in a transition. The transition has no useful behaviour associated with it so we use the DummyAction class. If necessary real behaviour can be inserted by creating an inner class just like we did with the initAction.

```
FSM atmFSM = new FSM();
atmFSM.addState(new AbstractFSMAction() {
  public void execute(FSMContext fsmc) {
    ...// desired behavior
  }
},"pincodeinvalid",null);
...// more states
FSMAction nothing = new DummyAction();
```

2.4.3. Problems and issues

While we no longer have to subclass FSMState, we still need to create FSMAction subclasses. However, these can be reused in various state transitions or even in other FSMs. A second issue may be performance. The transition lookup used to find the right transition for the right event is more expensive than a virtual method call. However, in our case this is not likely to be a very big problem since there will not be enough state transitions per second to notice the problem.

A second issue is that the FSMAction instances still need to be provided with a reference to the context that stores all the shared data. This is done by passing the context object as a parameter to the execute method:

`public void execute(FSMContext fsmc)`

Since, typically, this data is stored in a subclass of FSMContext, a typecast is needed. Apart from not being type safe, typecasts are also slower than normal referencing of variables.

Another problem is that creating a FSM now involves a lot of bookkeeping. ATMSimulator (now a subclass of FSMContext) consists of mostly static declarations of the states and transitions. Since we chose to use Java's inner class mechanism for creating the FSMAction subclasses, most of the ATMSimulator class consists of inner class declarations.

Effectively, we have created our own domain language where the various components form the language constructs. Unfortunately, a lot of bookkeeping is involved in using this language. We have to create subclasses of FSMAction, just to add behaviour to the system; we have to create component instances and link them together using method calls such as `addTransition`. For a more detailed discussion about the merits of this design solution, we refer to Van Gurp and Bosch (1999).

2.5. Version 5: further decoupling

2.5.1. New requirements

The goal of the fifth version of the ATMSimulator was to further reduce the dependencies on compile-time mechanisms. Version 4 still has a large static code block containing the specification of the ATM structure. This version addresses the following requirement:

- *Flexibility*. The solution in version 4 puts the entire ATMFSM in a single class. A lot of this code is made

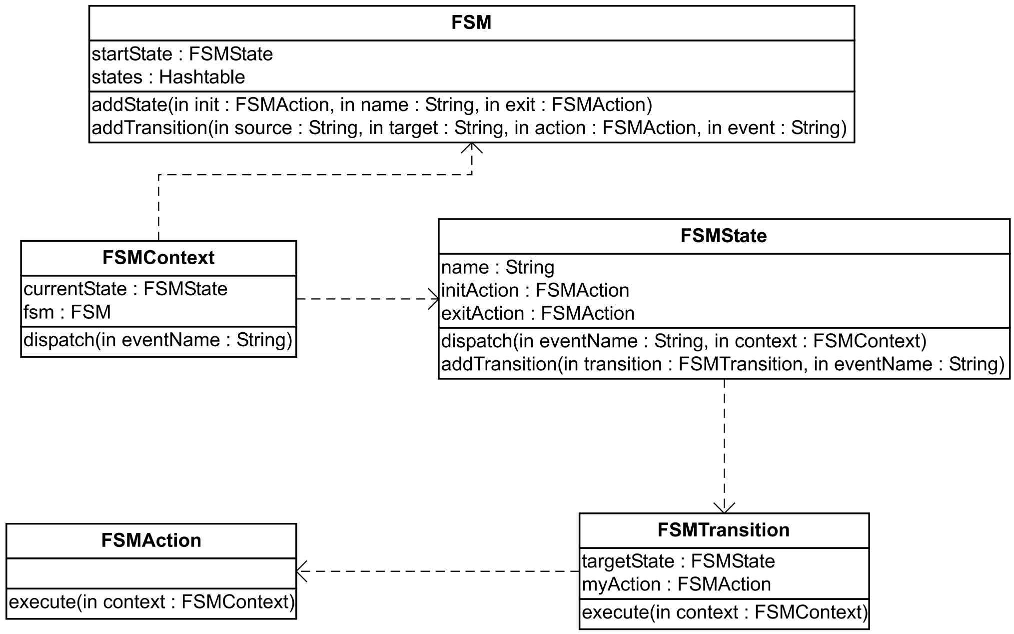

Fig. 6. Version 5: The new FSM class included.

static, which means that it cannot be changed at run-time and is difficult to maintain. In this version, we increase the flexibility by addressing this issue.

2.5.2. Changes

To address this we introduced a new class, FSM that can be used to create a FSM at run-time and contains information about the structure of a FSM. This separates the responsibility of storing the FSM structure from the more general FSM mechanisms of dispatching events. The new FSM class in Fig. 6 can be used in a blackbox fashion (i.e. it is not necessary to create subclasses of FSM). Example code that shows how to add states and transitions in the new version is listed below. atmFSM.addTransition("pincodeinvalid", "cardvalid", nothing, "validcard");

Typically, users create an instance of this class and use this instance to create the FSM by adding states and transitions. Then they create an FSMContext instance and parameterise it with the FSM. If necessary, more than one FSMContext instance can be created. If the FSM instance is changed, all existing FSMContexts are affected by it. Effectively, this separates the contextual information (i.e. the variables in FSMContext subclasses) from the structure (i.e. the states, events and transitions) and the behaviour (i.e. the FSMAction implementations).

While the changes to the FSM classes were minor, they had considerable consequences for the ATMSimulator specifics, which in the previous version consisted of a large static block of State declarations and addTransition method calls. In the new version all these calls had to be rewritten and were moved to the main method of the program (located in a class called ATMMain). The only remaining ATMSimulator specifics in this version of the system are the subclass of FSMContext containing all the variables used by the FSMAction implementations and the calls to the FSM instance in the main method that create the ATM state machine structure.

2.5.3. Problems and issues

We only addressed one issue identified for the previous version: the static declarations. So, all the other issues identified there also apply to this version.

2.6. Evolution of ATMSimulator

2.6.1. Important design decisions

In the development of the ATMSimulator, we can identify several important design decisions. Perhaps the single most important decision was to abandon inheritance in favour of delegation as a mechanism for creating new states. The most important design decisions and their effects are outlined in Table 1. As can be observed in this table, many of the decisions had system wide effects (e.g. decisions 1.1, 2.2, 3.2 and 4.1). In addition, some decisions effectively reversed decisionstaken earlier. The most notable example is decision 4.1 which effectively reversed 1.1. However, there are other examples: 2.2 reversed 1.3; 3.1 reversed 1.4 and 5.1 reversed 2.1.

2.6.2. Metrics

To compare the different versions we have collected a several metrics (Table 2). The metrics clearly show how

Table 1
Design decisions in the evolution of the ATMSimulator

Version	Decision		Effect on system
v1	1.1	Use the State pattern	For each state in a FSM, a subclass of State has to be created
	1.2	Put data in context class	Each event method in the State subclasses refers to the Context class to access data
	1.3	Make context a property of ATMState	The context is available to all State instances
	1.4	Use command line for UI	The code is littered with calls to System.in and System.out
v2	2.1	Make instances of State static	The keyword static needs to be put before instantiations of State subclasses
	2.2	Remove context property from ATM-State and use parameter in event method instead	All event methods need to be edited
v3	3.1	Create a GUI	A class is added to the system
	3.2	Replace System.in and System.out calls with calls to the GUI	All event methods need to be revised
	3.3	Apply the pipes and filters for communication between GUI and simulator	The changes needed in the event methods are relatively small
v4	4.1	Refactor the system to use delegation (Van Gurp and Bosch, 1999).	New classes are created that model the behaviour of states and transitions. All existing State subclasses are removed from the system.
	4.2	Use the command pattern to separate behaviour from structure	For each event method in the State subclasses, an inner class needs to be created that implements the FSMAction interface. An instance of such classes needs to be associated with the appropriate transition(s)
	4.3	Introduce state exit and entry events to the FSM model	The event dispatching mechanism needs to be changed to support this type of events
v5	5.1	Introduce factory classes for states and transitions	A new class is created. The initialisation code for FSMs can be made non static and becomes much simpler

the various design decisions affected the system. Some of the decisions had a positive effect on system complexity. We have drawn the following conclusions from the metrics:

- Overall system complexity (in terms of lines of code, lines of code per method, number of classes) has increased substantially from version 1 to 5.

Table 2
Metrics for the different versions

Versions:	v1	v2	v3	v4	v5
Number of packages	1	1	2	3	3
Number of (inner classes)	15	15	17	22	23
Number of functions	59	57	62	36	47
Non-commented source statements (ncss)	239	209	247	256	282
ncss/function	4.05	3.67	3.98	7.11	6
New (inner) classes	–	0	1	19	13
New functions	–	0	6	33	12
Removed (inner) classes	–	0	0	14	12

- Converting inheritance relations to delegation relations in version 4 was the most radical change.
- Version 5 has better modularisation than version 4. This is reflected in the decreased non-commented source statements (ncss) per function. Because modularisation also means increasing the number of modules (e.g. classes), the number of ncss is slightly larger than version 4.
- With the exception of version 2, each version has caused the total amount of ncss to increase.

However, not all changes are reflected in the metrics. In both versions 4 and 5 a considerable amount of existing code was rewritten (although we did use the copy/paste function a lot). In addition, the class refactorings between version 1 and 2 were considerable.

3. The minimal strategy

Based on the data in Tables 1 and 2, we can say that several of the design decisions would have been unrealistic in an industrial situation. Going from version 3 to

Table 3
Minimal strategy

Version	Decision		Alternative
v2	2.1	Make instances of State static	Unchanged
	2.2	Remove context property from ATMState and use parameter in event method instead	Use the array option described earlier to avoid having to move properties
v3	3.1	Create a GUI	Unchanged
	3.2	Replace System.in and System.out calls with calls to the GUI	Unchanged
	3.3	Apply the pipes and filters for communication between GUI and simulator	Unchanged
v4	4.1	Refactor the system to use delegation (Van Gurp and Bosch, 1999).	Change the ATM FSM to support disabling of the receipt option and other features that need to be supported
	4.2	Use the command pattern to separate behaviour from structure	Unchanged
	4.3	Introduce state exit and entry events to the FSM model	Add a stateEntry and stateExit method to the ATMState class and manually enforce that those methods are called when appropriate
v5	5.1	Introduce factory classes for states and transitions	Not needed

4, for instance, caused quite a few changes that affected the whole system. In large systems, consisting of a large amount of lines of code, such a change would effectively retire the old system and all the effort that went into it. The only reason the changes were feasible in our version was that our system is relatively small which enabled us to follow an optimal strategy for implementing the requirements. However, if we had followed a minimal strategy, the system would have looked differently. In this section, we outline what could have happened if we had followed the minimal strategy for evolving version 1. A summary of alternatives can be found in Table 3.

3.1. Version 1–2

The changes in this version consisted of moving class variables from ATMState to ATMContext and introducing a context parameter in all methods implementing state transitions. In an industrial sized system, this would have been considerably more work due to the larger number of classes and variables. An alternative might have been to use arrays that contain a variable for each instance of FSMContext. However, this would require a lot of changes as well and is ultimately more error prone.

3.2. Version 2–3

As pointed out before, the changes between these versions were designed in such a way that existing code was affected as little as possible. Even in our small version the better solution of using events has no option.

3.3. Version 3–4

As these were the most radical changes in the evolution of the simulator, they would probably not have been feasible in an industrial setting. The motivation for making the changes was that it would be nice to be able to make changes to the FSM structure to enable such features as dynamic disabling of the receipt function. However, as pointed out, the inheritance- based implementation is not very suitable for supporting this kind of dynamicity. In an industrial setting abandoning inheritance would simply be too much effort. A likely alternative would have been to identify the things that need to be configured at run-time (e.g. the receipt feature) and implement it either by making the FSM more complex (i.e. create transitions with and without the receipt functionality) or using some sort of boolean variable to control the behaviour.

3.4. Version 4–5

The last change was merely an optimisation of the design introduced in the previous version. Since that version would likely have never been created in the first place we do not provide an alternative solution here.

4. Analysis

The main goal of designing and implementing the various versions of the ATMSimulator was to observe and analyze what happens when a system is evolved as new requirements are added. By putting a strong emphasis on such requirements as flexibility, reusability and maintainability, our system began to show similar problems as those typically found in industrial cases.

4.1. Architectural drift

The initial version of the ATMSimulator was a relatively compact version. However, because of the design, maintainability and flexibility were less than ideal. We addressed these issues in the subsequent versions by changing the program structure; adding new classes; moving blocks of code around; etc. The design in version 5 still implements the same functionality as version 1. Yet, it is much larger and more complex. A lot of the new code is not functionality related but structure related. The added structure provides some additional flexibility over the first version. However, it also makes that version harder to understand. This may lead to architectural drift. Developers that do not fully understand the design may take sub-optimal decisions.

4.2. Vapourised design decisions

An example of a vapourized design decision in our system is the use of the pipes & filters architecture for communication with the GUI. This design decision only makes sense if you know that the simulator was originally equipped with a command line interface. Despite the fact that our system is limited to only five versions, most of the earlier design decisions vapourized. In a larger system there will be even more of these vapourized decisions.

4.3. Design erosion

Another issue is that version 5 shows some signs of design erosion, despite the fact that we tried to follow the optimal strategy. An example of this is the parameter of the execute method in each FSMAction implementation. This parameter passes the action a reference to the context that contains all of the shared variables. However, in our implementation we use a subclass of FSMContext that contains these variables. Consequently, all actions must perform a typecast on the context parameter to get access to these variables. A second sign of design erosion is the solution used to connect the GUI to the state machine. The pipes and filters solution we chose was a direct result from the fact that the first version was command line based. Since we tried to preserve much of the functionality in this version, we had to somehow duplicate this type of interactive behaviour. Our solution consisted of connecting a text field to a pipe that on the other side was connected to the so-called BufferedReader that functions in a similar way as the input from the console we used in the first version. While this allowed us to preserve much of the code, an event-based approach would have been more natural if we had build version 5 directly.

All these characteristics of the final version are a result of design decisions taken in earlier versions. Because of changes in the requirements, these decisions can no longer be considered as optimal for version 5. Consequently, version 5 is not the optimal design for the requirements we specified for it. Yet, constructing an optimal system would mean abandoning much of the code we already wrote in earlier versions. These problems are even worse in the version of the system we presented in Section 3, since this version contains many ‘quick fixes’.

Arguably, in our prototype throwing away large parts of the code is not a very big issue (because of its small size). Our intention is to illustrate to the reader that this sort of problems also occur in large industrial systems that evolve throughout the years. Each design decision in itself can be seen as valid. However, when considered all at once there may very well be a more optimal system. Because of the legacy of existing code, which in an industrial setting often represents an investment of many person years, this is no option, however. In Section 3 we discussed alternative implementations for our simulator that would have been more likely in an industrial setting. The quick and dirty fixes discussed in this section clearly do not contribute to the clarity of the code. Using such solutions as global arrays to prevent adding a parameter to a method, solve the problem at hand but at the same time contribute to the erosion of the design.

4.4. Accumulated design decisions

A related issue is that of hardwired design decisions. In the ATMSimulator, we had a major restructuring of the code between versions 3 and 4. This was caused by our decision to abandon the State pattern, adopted in version 1. This earlier decision had an enormous impact on the code structure (see Table 2). Undoing it required quite a lot of effort and might not have been feasible in a larger project with hundreds of states and events. It also caused us to reconsider other decisions such as decision 2.1 in Table 1.

4.5. Limitations of the OO paradigm

The changes between each version aimed to resolve a particular issue in the previous version. One could argue that version 5 addresses all issues we encountered during development. However, we already showed that version 5 may not be the most optimal system, despite the optimal design strategy we applied. We suspect that many of the solutions we presented are workarounds for problems with the OO paradigm.

- *Inheritance.* The reason we moved from an inheritance-based to a delegation-based solution in version 4 was that we needed run-time flexibility. The inheritance-based solution was more compact (i.e. was a better expression of the functionality) however inheritance makes it impossible to meet the run-time flexibility requirement so we needed to work around it.
- *Typecasting.* From version 2, the FSMContext no longer was a property of the state objects. Consequently, when performing a state transition, references to the context object needed to be passed as a parameter. In version 4 and later, we use subclasses of FSMContext to model the context. The FSMAction interface defines an FSMContext parameter, however. So, consequently we have to use type casting to resolve this. This is a known issue with the OO paradigm and there is a good solution for it: parameterised classes. However, this is not supported in Java currently.
- *Encapsulation.* The OO paradigm prescribes us to encapsulate data into objects. However, in our ATMSimulator the quality requirements forced us to centralize data in the ATMContext class (and later subclasses of FSMContext). To reduce memory overhead, we had to apply the Flyweight pattern. Because of the above we violated Demeter's law (Lieberherr and Holland, 1989) that prescribes that only calls to objects which are class variables in which the call originates and calls to objects that are passed as a parameter of the method from which the call originates, are legal.

4.6. Optimal vs. minimal strategy

As pointed out before, several of the decisions in Table 1 would not have been feasible in a larger system. This kind of decisions is typical for what we call an optimal strategy for implementing requirements. In Section 3 we outlined some alternatives for some of those decisions. These alternatives have in common that they address the immediate need (e.g. run-time flexibility) while minimizing impact on the system. The short-term advantage is that it speeds up development. However, in the long-term this type of decisions becomes an obstacle for further development. However, even the optimal strategy does not lead to an optimal design. It just delays inevitable problems like design erosion and architectural drift.

4.7. Lessons learned

Based on our experiences with the development of the five versions of the ATMSimulator, we can draw some conclusions.

- Some conceptually simple design decisions have enormous consequences for the code. The decision to abandon inheritance as a mechanism for creating new states in version 4, for instance, caused a lot of code to be moved around.
- The differences between the initial version and the final version are considerable. Without the knowledge of the in-between versions, it is hard to deduce why the system looks the way it does.
- In none of the versions, a quantification of the quality attributes was the driving force behind the changes. Instead, in each case a particular usage or change scenario drove the changes.
- Our requirement for run-time flexibility caused us to use design patterns such as the Flyweight pattern and the Command pattern. While these commonly used design solutions work, the result can seem overly complex. In the first version, behaviour of a transition could be changed by changing a method, in the final version, the FSMAction class needs to be sub classed. The subclass must define an execute method. Then an instance of the newly created subclass needs to be created and inserted into the transition. While Java provides some syntactic sugar (e.g. inner classes), the whole procedure seems awkward.
- A lot of the code refactorings in-between versions involve a lot of more or less mechanical changes (e.g. cutting and pasting lines of code). This suggests that some of these refactorings can be automated as for instance is done for some refactorings in Roberts et al. (1997).
- Later design decisions become more difficult because the earlier design decisions have to be taken into account. Even in our small prototype, we had to deal with the legacy of the first few versions when going from version 4 to 5. This caused us to move around a lot of code.

4.8. Research issues

To be able to prevent and counter design erosion, a lot of research is needed. We have identified a number of issues that we feel need to be addressed. Some of these issues are already the topic of existing research. However, this research has not yet brought us to the point where we can prevent design erosion.

- *Separation of concerns.* There is a lot of ongoing research in this area (e.g. Kiczalez et al., 1997; Lieberherr, 1996; Tarr et al., 1999). However, we have the impression that most of this research focuses on isolating smaller pieces of code rather than larger architectural components. It is unclear if and how such techniques will scale when used in conjunction with very large industrial systems. So far there is hardly any case study material to confirm the effectiveness of these techniques in larger systems.

- *Expressiveness of representations.* Related to the previous issue is the representations used to model a system. We have experienced that more often than not the source code is the documentation. Consequently, many of the concepts used during the design phase are represented in an implicit fashion. This causes serious maintenance issues since maintainers will have to reconstruct the design from the source code before they can change it.
- *Refactoring.* There has been some promising research into code refactoring [most notably (Fowler et al., 1999; Roberts et al., 1997)]. However, more advanced, preferably automated, refactorings would be useful.
- *Methodology.* As pointed out in this paper, most existing development methods are flawed because they iteratively accumulate design decisions. Since it is inevitable that requirements change over time, it is also inevitable that eventually design erosion occurs (because some of the earlier decisions become invalid). Current research focuses on fighting the symptoms (i.e. design erosion) rather than the problems (i.e. the previous topics). New methodologies such as extreme programming (Beck, 1999) address this by adopting a stepwise refinement strategy with frequent releases. However, there are issues with respect to, among others, planning and cost management of projects using such methods.

4.9. Research limitations

A limitation of our study that we have used is that the example we used is relatively small when compared to industrial cases such as we presented in for instance Bengtsson and Bosch (1998) and Bosch et al. (1999). The reasons we chose to us this small example are:

- We can control the evolution of the software system. We wanted to demonstrate the difference between following the minimal effort strategy and the optimal architecture strategy. In an industrial setting the conditions under which an architecture evolves are hard to control and e.g. cost factors will influence the result.
- It is small enough to discuss in detail in the context of a paper. A full blown industrial case is too large to discuss in full with the level of detail needed for this paper.
- Despite being rather small, the FSM framework that we used as the basis for this paper has some industrial relevance. The original paper that described the finite state machine architecture (Van Gurp and Bosch, 1999) has had some positive response from people in industry. This suggests that the design of the framework is not unreasonable.
- We felt that the conclusions of our experiences with this case were important enough to publish despite the fact that our small example can only provide a limited amount of evidence for these claims. Of course additional case studies are needed to further validate our conclusions and to learn more about design erosion.

We are currently working on an industrial case study that addresses some of the limitations of this study. This case study focuses on how concerns are separated in two software companies and what design decisions are important in doing so.

5. Conclusion

In this paper we have evaluated an extensive example of evolutionary design to assess what happens to a system during evolution. The example clearly demonstrates how design erosion works. Design decisions taken early in the evolution of a system may conflict with requirements that need to be incorporated later in the evolution. In the example, we reversed several of such decisions. However, in large industrial systems such a thing is often infeasible due to the radical, system wide impact of such changes.

In the analysis of our design efforts we have found evidence of architectural drift, vapourized design decisions and design erosion. Causes we identified for these problems ranged from the accumulation of multiple design decisions (i.e. certain design decisions were taken because of earlier design decisions, even if these were wrong decisions) to limitations of the OO paradigm. An important conclusion is that even an optimal design strategy (i.e. no compromises with e.g. cost are made) for the design phase does not deliver an optimal design. The reason for this is the changes in requirements that may occur in later evolution cycles. Such changes may cause design decisions taken earlier to be less optimal.

5.1. Future work

In our analysis of the case study, we highlighted several issues. One of them, limitations of the OO paradigm, will form the starting point for our future research. We intend to explore alternatives and extensions to the OO paradigm as possible solutions to the issue of design erosion. It appears that using the OO paradigm, some important concerns are mixed. Untangling those concerns may be the key to addressing at least some of the issues identified in this paper. We are currently finishing a case study in two local Small and Medium sized

Enterprises (SME's) that explores what concerns are important in software companies.

A second issue that we intend to explore is that of the design method. It seems that the current practice of software development is to create a design in advance. However, as noted in Section 1 this conflicts with the iterative nature of many development methods. New requirements are constantly added to the system and as our case study demonstrates they often conflict with design decisions taken in earlier iterations or in the design phase. We believe such conflicts are the primary cause for the phenomena of design erosion.

Finally, the issues we highlight in Section 4.9 need to be addressed. Partially, the case study we mentioned earlier will serve this purpose. However, the focus of that case study is different from the study presented in this paper and additional studies that address the weaknesses of this paper are necessary to provide additional evidence for the claims presented in this paper.

References

Beck, K., 1999. Extreme Programming Explained. Addison-Wesley, Reading, MA.

Bengtsson, P., Bosch, J., 1998. Scenario-based software architecture reengineering. In: Proceedings of the 5th International Conference on Software Reuse, pp. 308–317.

Bosch, J., 1999. Evolution and composition of reusable assets in product-line architectures: a case study. In: Proceedings of the First Working IFIP Conference on Software Architecture.

Bosch, J., 2000. Design and use of software architectures adopting and evolving a product-line approach. Addison-Wesley, Reading, MA.

Bosch, J., Molin, P., Matsson, M., Bengtsson, P., 1999. Object oriented frameworks – problems and experiences. In: Fayad, M.E., Schmidt, D.C., Johnson, R.E. (Eds.), Building Application Frameworks. Wiley, New York.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1996. Pattern-oriented Software Architecture: A System of Patterns. Wiley, New York.

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., 1999. Refactoring – Improving the Design of Existing Code. Addison-Wesley, Reading, MA.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, Reading, MA.

Harrison, W., Ossher, H., 1993. Subject-oriented programming (a critique of pure objects). In: Proceedings of OOPSLA '93.

Jaktman, C.B., Leaney, J., Liu, M., 1999. Structural analysis of the software architecture – a maintenance assessment case study. In: Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1).

Kiczalez, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J., 1997. Aspect oriented programming. In: Proceedings of ECOOP 1997.

Lieberherr, K., 1996. Adaptive Object-oriented Software – The Demeter Method. PWS Publishing company.

Lieberherr, K.J., Holland, I.M., 1989. Assuring good style for object oriented programs. IEEE Software, 38–48.

Naur, P., Randell, B., 1969. Software Engineering: Report on 1968 NATO Conference, NATO, 1969.

Parnas, D.L., 1994. Software aging. In: Proceedings of ICSE 1994.

Perry, D.E., Wolf, A.L., 1992. Foundations for the study of software architecture. In: ACM SIGSOFT Software Engineering Notes, vol. 17, No. 4.

Roberts, D., Brant, J., Johnson, R., 1997. A refactoring tool for smalltalk. Theory and Practice of Object Systems 3 (4), 253–263.

Svahnberg, M., Bosch, J., 1999. Characterizing evolution in product-line architectures. In: Proceedings of the IASTED 3rd International Conference on Software Engineering and Applications.

Tarr, P., Ossher, H., Harrison, W., 1999. N Degrees of separation: multi-dimensional separation of concerns. In: Proceedings of ICSE'99.

Van Gurp, J., Bosch, J., 1999. On the implementation of finite state machines. In: Proceedings of the 3rd Annual IASTED International Conference Software Engineering and Applications.

Jilles van Gurp is a Ph.D student at the University of Groningen. After obtaining an M.Sc at the University of Utrecht, he moved to Sweden to start work on his Ph.D at the University of Karlskrona/Ronneby under the supervision of Jan Bosch. In 2000, he moved back to the Netherlands with Jan Bosch and is currently continuing the work on his Ph.D at the University of Groningen.

Jan Bosch is a professor of software engineering at the University of Groningen, The Netherlands, where he heads the software engineering research group. He received an M.Sc degree from the University of Twente, The Netherlands, and a Ph.D degree from Lund University, Sweden. His research activities include software architecture design, software product lines, object-oriented frameworks and component-oriented programming. He is the author of a book "Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach" published by Pearson Education (Addison-Wesley & ACM Press), (co-)editor of three volumes in the Springer LNCS series and has (co-)authored more than 50 refereed journal and conference publications. He has organised numerous workshops, served on many programme committees, including the ICSR'6, CSMR'2000, ECBS'2000, GCSE, SPLC and TOOLS conferences and is member of the steering groups of the GCSE and WICSA conferences. He is the PC co-chair of the 3rd IFIP (IEEE) Working Conference on Software Architecture (WICSA-3).