Adaptive Strategy Design Pattern

Olivier Aubert – Olivier.Aubert@enst-bretagne.fr,
Antoine Beugnard – Antoine.Beugnard@enst-bretagne.fr
Laboratoire d'Informatique des Télécommunications, ENST Bretagne, France

June 25, 2001

Abstract

According to a definition found in the summary of the Workshop on Adaptable and Adaptive Software[LL95]: *"A program is called adaptive if it changes its behaviour automatically according to its context"*. Within this context, we restrict our research domain to the automatic runtime adaptation of existing behaviours.

In this paper, we propose an Adaptive Strategy Design Pattern that can be used to analyze or design self-adaptive systems. It makes the significant components usually involved in a self-adaptive system explicit, and studies their interactions. We show how the components participate in the adaptation process, and characterize some of their properties.

1 Introduction

Self-adaptive software has recently seen a surge of interest in the computer science community. Advances in modeling, hardware performance and introspection make it seem more plausible than before. It has previously been used in specialized fields such as load-balancing[RSZ87, Har98], scheduling[MP90], protocols[HMS99] or mobile computing[SA00, FDBC99]. However, its more specific study is the subject of recent projects[Lad98, GH91, BN99, Lad99, KBE99, OGT+99] and conferences[LL95].

According to a definition found in the summary of the Workshop on Adaptable and Adaptive Software[LL95]: *"A program is called adaptive if it changes its behaviour automatically according to its context."*

Even within this definition, we can find some room for variants. We will further restrict our research domain by stating that we would like to provide a model for automatic runtime adaptation of existing behaviours.

In this Design Pattern, we will try to explicit the significant components usually involved in a self-adaptive system and study their interactions. We will see how the components participate in the adaptation process, and characterize some of their properties.

2 The *Adaptive Strategy* Design Pattern

2.1 Intent

Define a self-adaptive strategy, exposing to the client a single strategy referencing the best available concrete strategy, only requiring from the client an access to the environment information that can be used to choose the best strategy.

2.2 Motivation

Mobile systems can greatly benefit from adaptivity. Consider the example of a mobile device needing to access data located on fixed servers. The mobile device may be disconnected from the network and thus unable to access the data. If a network access is available, the device can be weakly or strongly connected, depending on the quality of the connection.

When strongly connected, the device can use a standard transmission mode, e.g. a standard file sharing protocol. If the device is going to be disconnected, a copy of the data is made locally so that remote access is not needed. Various systems[SKK+90, SA00] implement all or part of this functionality, at different levels. Three different behaviours are thus used: one for the strongly connected mode, one for the low-bandwidth mode and one for the disconnected mode. The mobile system has to choose between them, basing its decision on the observation of the context.

The adaptation process involves four main steps. First, the system has to be able to **monitor its environment**, either by observation (for instance, bandwidth evaluation) or by notification (for instance, the system is notified when the mobile device is going to be disconnected). Based on this information, the system **makes a decision**, i.e. selects, if necessary, a new behaviour. If it decides to replace the current behaviour, it may have to **handle the transition** between the old and the new behaviour, which may involve transmitting some state information. In our example, it can be the queue of messages waiting to be sent. Once the new behaviour is ready to be activated, the system can eventually **enact the changes** and validate the new configuration.

The *Strategy* Design Pattern[GHJV95] already provides a repository of algorithms, and allows them to be interchangeable. However, the pattern leaves the choice of the right strategy up to the client. The *Strategy* pattern is thus necessary, but not sufficient, to deal with self-adaptation. Building on it, we identify other components involved in the adaptation process, namely an InformationGateway offering monitoring and observation capabilities, a Controller that makes the decision, a StateAdapter dedicated to the adaptation and transmission of information between strategies during the transition phase, and the Adaptive Strategy that glues the components together and provides an entry point for the clients.

Figure 1 on the following page shows a class model for our simplified example.

The aim of the proposed Design Pattern is to formalize the components used in the adaptation process and discuss some implementation issues.

2.3 Applicability

Use the *Adaptive Strategy* Design Pattern when:

- different versions of an algorithm are available, and we cannot decide until runtime which version is best suited for the task.
- differently stated, you are creating an autonomous system (or agent) whose behaviour can change over time according to external and internal conditions.
- the method used to select an algorithm is well-defined for a case, and the client should not have to worry about it, so you want to provide an adaptive component which should be simple to use and reuse.

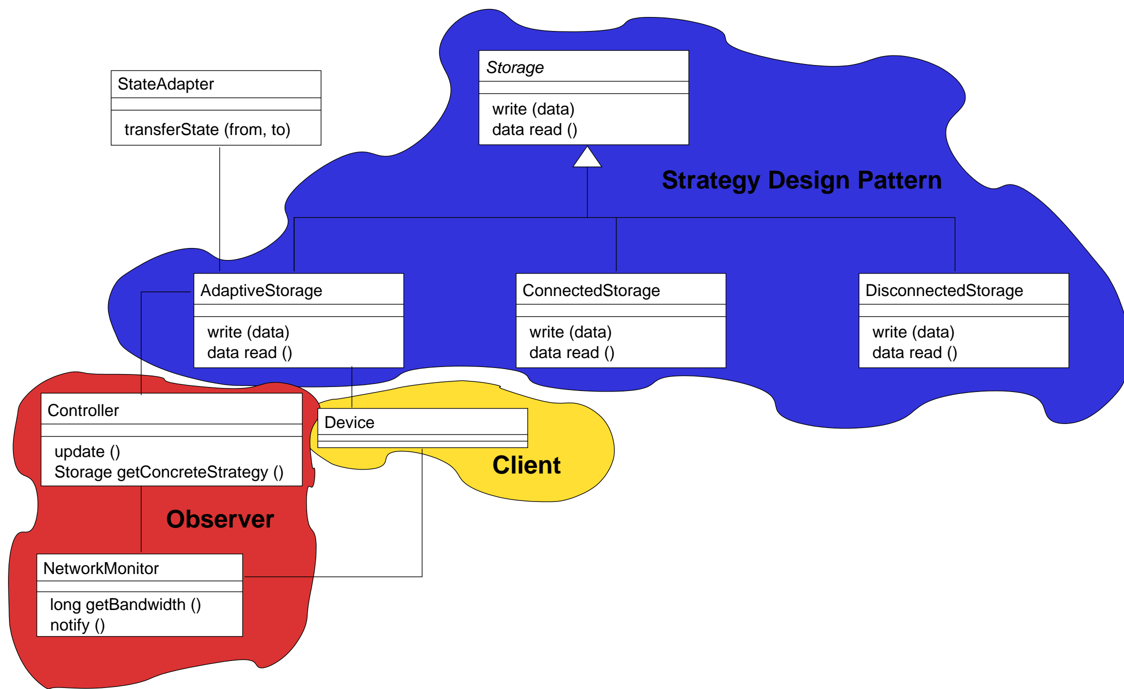

Figure 1: Example class model

2.4 Structure

Figure 2 on the next page shows the general structure of the Design Pattern[1].

2.5 Participants

This pattern is built on the *Strategy* Design Pattern (behavioural), the *Facade* Design Pattern (structural), and the *Observer* Design Pattern.

Client is the object using the Strategy services. It provides information needed by the Strategy implementations.

Strategy declares an interface common to all supported algorithms. Client uses this interface to call the algorithm defined by a ConcreteStrategy.

ConcreteStrategy implements the concrete algorithm using the Strategy interface. Each ConcreteStrategy may have an associated state, depending on the nature of the implemented service.

Adaptive Strategy is the main access point for the clients of the component. It provides a single entry point, a *Facade*, for the different Strategy functionalities.

The fact that Adaptive Strategy is itself designed as a Strategy leads to a simplification of the model, and exemplifies the decoupling of the mechanism implemented by the Strategy from the adaptation mechanism. Ideally, we should be able to transparently replace the

[1]The class diagram and collaboration diagram in figures 2 on the following page and 3 on page 5 use the UML notation, but this does not mean that direct application should be derived from them.

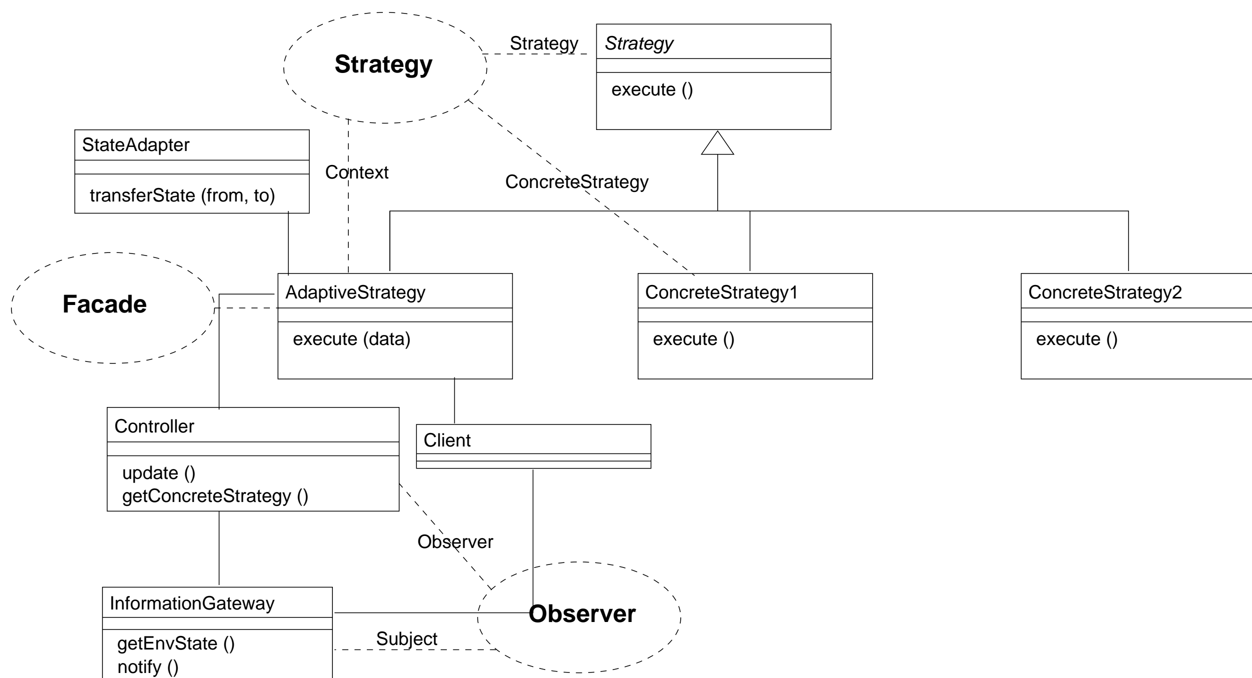

Figure 2: Class diagram for the Adaptive Strategy Design Pattern

Adaptive Strategy by one of the ConcreteStrategy, as a form of optimization by specialization for instance.

Controller has the difficult task of choosing the best algorithm, using information gathered from the InformationGateway. It is called by the Adaptive Strategy object.

InformationGateway represents the gateway through which the Controller gets the necessary information to make its decision. It provides access methods to environment information. Thus it constitutes the glue that is necessary to adapt a generic Adaptive Strategy (AdaptiveStorage for instance) to a specific system.

- It defines an interface that lets the Controller access runtime or historical environmental parameters.
- It may also provide active capabilities, which can trigger actions according to environment changes by notifying other components. This functionality can be implemented with the help of the *Observer* Design Pattern.

StateAdapter handles if necessary the transition between two strategies that need to have some state information exchanged.

2.6 Collaborations

Two types of adaptation can be distinguished[BMN+00]. *On action* adaptation, as shown in figure 3 on the following page, occurs when a method of the strategy is called. *On change* adaptation, presented in figure 4 on page 6, is triggered by a change in the environment, independently from

the invocation of the methods. Our example of data access from a mobile device uses an *on change* adaptation.

2.6.1 *On action* adaptation

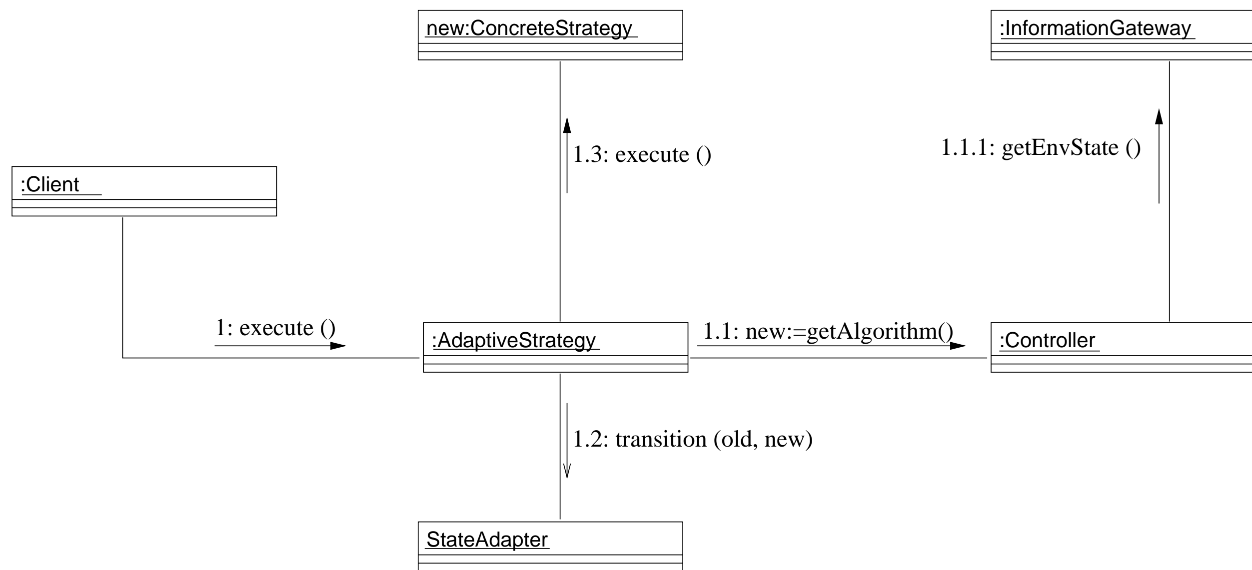

Figure 3: Collaboration diagram for *on action* adaptation.

Figure 3 on this page shows a possible collaboration[2] between the various constituents of our system in the case of an *on action* adaptation.

Adaptive Strategy is the main entry point for the clients (the Context in the original *Strategy* Design Pattern). It receives a request (*1*) for an operation, and asks (*1.1*) the Controller which ConcreteStrategy is the most appropriate to process that request. The Controller chooses the best strategy based on the information provided (*1.1.1*) by the InformationGateway and informs the Adaptive Strategy about its choice. If there is a change of strategies, and both strategies share some state information, the StateAdapter is invoked to manage the adaptation and transfer (*1.2*) of the pertinent information from the old strategy to the new one. Once the transition is achieved, the Adaptive Strategy can then forward the request (*1.3*) to the appropriate ConcreteStrategy.

2.6.2 *On change* adaptation

Figure 4 on the next page shows a collaboration between the entities of our Design Pattern in the case of an *on change* adaptation. For this type of adaptation, the *request processing*, indexed as A evolves independently from the *adaptation process*, indexed as B.

During the adaptation process, the InformationGateway notifies (*B.1*) the Controller that the adaptation parameters have changed. The Controller then gets (*B.1.1*) the necessary environmental information, makes its decision, and informs (*B.1.2*) the Adaptive Strategy if the ConcreteStrategy has to be replaced. The Adaptive Strategy can invoke (*B.1.3*) the StateAdapter to adapt and transfer information between the old and the new strategy if necessary.

[2]The link between InformationGateway and Client is not shown

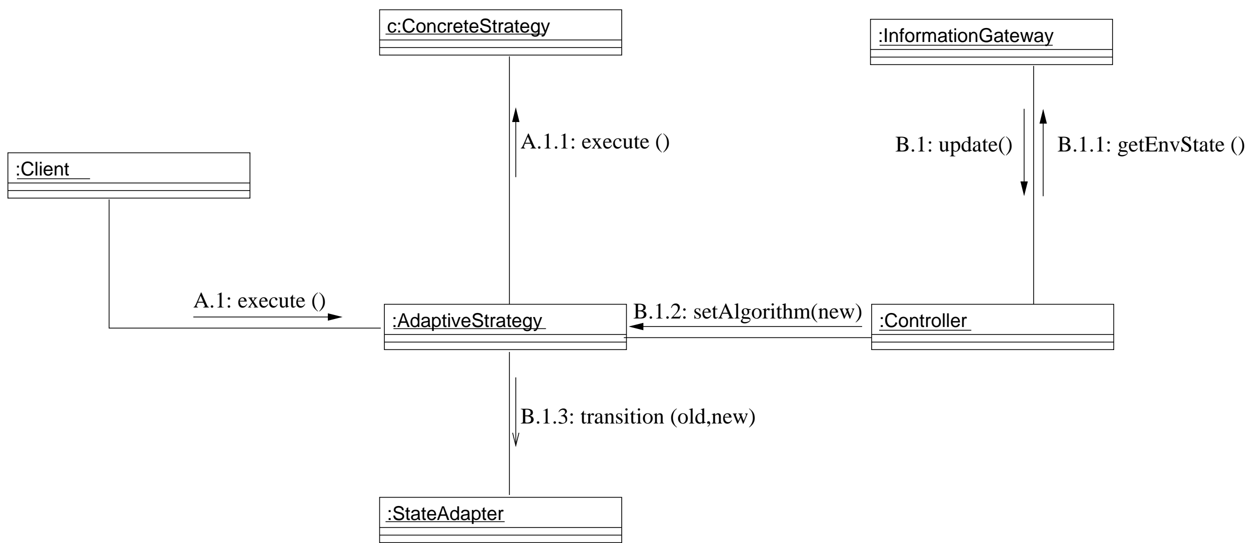

Figure 4: Collaboration diagram for *on change* adaptation - A and B sequences are independant.

On the other hand, requests (*A.1*) for operations are managed by the Adaptive Strategy, which directly forwards (*A.1.1*) them to the ConcreteStrategy.

2.7 Consequences

The *Adaptive Strategy* Pattern has the following consequences:

- *Hiding details from clients.* Our Design Pattern provides a unique interface to invoke the best algorithm, saving the client the trouble of choosing it. It benefits from a more precise knowledge of execution conditions so that it is able to choose the most appropriate available strategy.
- *Separation of concernes: distinguishing the adaptation process from the execution process.* Our Design Pattern promotes a clear separation between these issues which leads to more clarity in the comprehension of their mechanisms and interactions.
- *Make the StateAdapter explicit.* The transition phase is often neglected in adaptive systems. The StateAdapter reifies this important aspect in the design of an adaptive system.
- *Consequences on extensibility.* The addition of a new strategy is on the one hand made easier by the reification. On the other hand, the Controller has to be informed of all the available ConcreteStrategies and of their respective charateristics, in order to be able to choose the most appropriate among them. Thus the implementation of the Controller, discussed in section 2.8, may in some cases hinder the addition of new strategies.

2.8 Implementation

2.8.1 Adaptation process

- The process of dynamic adaptation can be divided into four basic steps, inspired from [OGT+99]:

  - evaluate and monitor observations (InformationGateway)
  - make a decision (Controller)
  - handle transitions (StateAdapter)
  - enact changes (Adaptive Strategy)

  Each step involves more particularly a part of the Design Pattern.

- The evaluation and monitoring part achieved by the InformationGateway has to be carefully designed. It can have a great impact on the overall performance of the system, if it is too intrusive. Various approaches have been studied[SA00, Lad99, BOK+99], involving in particular a hierarchical organisation of observations. Both pull and push approaches can be used.

  The information accessible via the InformationGateway can be either from the past (historical information), from the present (current state information), or from the future (in the case of prediction information).

- Different approaches can be used to implement the Controller. They have in particular an impact on the extensibility of the system. Using an ad-hoc function implies the modification of the Controller when adding a new strategy. Using cost functions[EKC98], associated to each strategy, could help to alleviate this problem. Other approaches include heuristics, neural network, simulator[SS98], state automaton[SA00], etc.

- How can we guarantee that the decisions of the Controller cover the whole range of situations? We have to make sure that every case is taken into account. One of the ways to ensure this is to define a default algorithm which will be valid in all cases, even if suboptimal.

- The Controller is the concretization of the goal of our system. An adaptive system has a goal: maintaining a connection, optimizing bandwidth use, etc. The implementation of the choice algorithm of the Controller is a translation of this goal into code. A change in the goal is then translated into either a change in the choice algorithm, or a change in its parameters.

  An analogy can be made with a complex system with many control knobs, whose complexity is hidden by the Controller which provides only one big control knob (for instance in a web cache case, the saved bandwidth vs. access latency, although the real situation is more complex).

- Depending on the statefulness of the strategies, the transition phase from one strategy to another can involve transmitting state information from the old strategy to the new one. The StateAdapter is in charge of the adaptation and transmission of the state information.

- Once the new strategy has been chosen and the transition handled, the Adaptive Strategy can validate the adaptation.

  Two important issues must be taken into account. First, means should be provided by the Adaptive Strategy in order to prevent service requests to be lost or mismanaged during the transition phase.

  Second, the stability of the system must be ensured, even in the case of quickly changing conditions. Stability is conventionally achieved in hardware adaptive devices via a hysteresis mechanism[KP89].

2.8.2 Adaptation classification

As mentioned in section 2.6, [BMN+00] identifies two types of adaptation:

- *on change* adaptation is triggered by a change in the environment, typically notified by the InformationGateway component. In this case, a reference to the concrete strategy is usually kept and invoked when necessary, using a *Proxy* Design Pattern. In our example, a change in the network conditions can cause a change in our strategy of accessing data.

- *on action* adaptation occurs at the strategy invocation time. This kind of adaptation is especially used when one of the decision criteria depends on some property of the parameters of the algorithm. For instance, an adaptive sorting algorithm depends on the properties of the data to be sorted.

The desired behavior may involve both types of adaptation. For instance, in the mobile storage example, we may have an *on change* adaptation handling the variations in bandwidth, and an *on action* adaptation intended to optionally compress the data, according to its type.

The combination of both types of adaptation can be achieved in two ways. The first one consists of simply implementing the necessary means as a set of ad-hoc mechanisms in the Controller. The Controller will then be activated by the InformationGateway for an *on change* adaptation, as well as by the client at each invocation for an *on action* adaptation.

Another method would be to apply our Design Pattern to the Controller itself. Indeed, the Controller can be built as an adaptive strategy, which would switch its selection algorithm according to the environment. An *on change* adaptation for the Controller has to choose between the concrete ConnectedController and DisconnectedController. These concrete strategies are then used as the Controller for the other, *on action*, adaptation.

2.8.3 Strategy classification

A simple distinction would be to distinguish between *stateless* and *stateful* strategies. The statelessness of a strategy induces many simplifications in the strategy change policy discussed below.

For instance, various sorting strategies do not need to share state information. On the other hand, a concurrency strategy[KC99], featuring as alternatives a ThreadPool, Single-Threaded and Thread-Per-Connection strategies, needs to transmit the enqueued incoming connections waiting to be handled from a strategy to another.

2.8.4 Strategy change policy

Once the Controller has decided that the current strategy should be replaced comes the question of how to handle the transition from the old strategy to the new one.

In the case of stateful strategies, if we have to keep some state information from one strategy to another, the StateAdapter adapts and transmits the state information.

MolèNE[SA00] uses a component named *state adapter* which allows the transfer of information between *Implementation* objects. If the information is sufficiently similar, this service uses a simple *Memento* Design Pattern. If the information is different, the *state adapter* has to select significant information from the source Implementation to be given to the new Implementation.

The 2K[HKC+99] distributed operating system relies on a CORBA ORB called dynamicTAO[RKC99]. It handles transitions using a dedicated component called *ComponentConfigurator*[KC99], which also uses the *Memento* Design Pattern.

2.9 Sample Code

We will here describe mainly interfaces for our mobile storage access, which features an *on change* adaptation. An abstract class defines the Storage interface:

```
public abstract class Storage {
    public abstract void write (Data d);
    public abstract Data read ();
}
```

Three concrete classes are derived thereof. One for each mode of operation (connected or disconnected) and the AdaptiveStorage class that will act as a *Proxy* for the other concrete strategies.

```
public class ConnectedStorage extends Storage {
  public void write (Data d) { ... };
  public Data read () { ... };
}
public class DisconnectedStorage extends Storage { ... }
public class AdaptiveStorage extends Storage { ... }
```

NetworkMonitor monitors the network bandwidth and notifies the Controller when an important change occurs.

```
public class NetworkMonitor {
  public long getBandwidth () {...};
  public void notify () {...};
}
```

Controller is an *Observer* of InformationGateway. When notified of changes, it selects the most appropriate concrete strategy, and informs the AdaptiveStorage.

```
public class Controller {
  public void update () {...};
  public Storage getConcreteStrategy () {...};
}
```

StateAdapter transfers state information, for instance the queue of messages not yet sent, from the current strategy to the strategy that will be activated.

```
public class StateAdapter {
  public void transferState (Storage from, Storage to) {...};
}
```

The application only has to invoke AdaptiveStorage methods to access the data in the most appropriate way.

```
AdaptiveStorage st;
Data d;

st.read (d);
st.write (d);
```

2.10 Known Uses

The *Adaptive Strategy* Design Pattern can be used to analyze existing adaptive systems.

The ISTORE Project[BOK+99] aims to provide self-adaptive data storage, by combining hardware and software elements. Their system couples LEGO-like plug-and-play hardware components, having active monitoring capabilities, with a generic framework for constructing adaptive software. The decoupling between observation, control and adaptation is clear. However, transition's handling is not explicit.

Operating systems can also benefit from adaptive features, in many aspects. The Synthesis[MP90] operating system introduces fine-grained adaptive scheduling, with a design inspired by the hardware phase-locked-loop. The VINO[SS98] operating system provides a general framework, based on the notion of *grafts* to allow self-adaptation of various mechanisms like memory paging, disk wait, interrupt latency, etc. Their implementation also decouples observation, control and execution.

In the network domain, Adaptive Protocol[HMS99] also points out the need for a clear distinction between mechanism and policy. In their model, functions can be dynamically inserted and removed in response to changes in the environment.

Various approaches to adaptivity, from language approaches[BMN+00] to middleware ones[SA00, HKC+99], operating systems[SS98] or more generic approaches, like the Viable Systems Model[HK00], feature some or all of the aspects mentionned in this Design Pattern.

2.11 Related Patterns

- *Strategy*
- *Observer*
- *Facade*

This pattern is primarily an extension of the *Strategy* Design Pattern.

3 Related work

The specific issues of self-adaptivity have been largely discussed in the domain of control theory applied to industrial processes. The inspiration of control theory to the design of software systems has already been studied in [OGT+99, Lad99, KBE99], but cover a wide range of adaptation types, from parametric adaptation to architecture reconfiguration.

[BN99] has an approach similar to ours of analyzing self-adaptive systems through the composition of Design Patterns. The composition of the *Strategy* and *Observer* Design Patterns is studied in depth, and it also applies to behaviour changes initiated by other objects. The separation of the different entities taking part in the adaptation process is not pushed as far as necessary, though.

4 Conclusion

In this paper we have identified the different entities composing a self-adaptive system, and we have shown their interaction during the four steps of the adaptation process: evaluation and monitoring, decision-making, transition-handling, and activation of changes.

The usefulness of this Design Pattern is twofold. On the one hand, it provides a support for the analysis of existing self-adaptive systems. On the other hand, it can be used during the design

of a self-adaptive system to identify the issues and the needed tools. A concrete application of the *Adaptive Strategy* Design Pattern is currently being developped in an adaptive web cache system.

The mechanism of state adaptation will be the subject of a more precise study leading to a *State Adapter* Design Pattern.

5 Acknowledgements

Maria Teresa-Segarra has provided substantial comments and reviews of this pattern and the careful and detailed feedback from our KoalaPLoP shepherd, Gustavo Rossi, led to significant improvements. Thanks to the KoalaPLoP attendees, and especially to Charles Herring, for the feedback and the fruitful discussions.

References

[BMN+00] P. Boinot, R. Marlet, J. Noyé, G. Muller, and C. Consel. A declarative approach for designing and developing adaptive components. In *15th IEEE International Conference on Automated Software Engineering*, 2000.

[BN99] Mathias Braux and Jacques Noyé. Changement dynamique de comportement par composition de schémas de conception. In *Langages et Modèles à Objets LMO'99*, january 1999.

[BOK+99] A. Brown, D. Oppenheimer, K. Keeton, R. Thomas, J. Kubiatowicz, and D.A. Patterson. Istore: Introspective storage for data-intensive network services. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII), Rio Rico, Arizona*, march 1999.

[EKC98] Michael D. Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP '98, the 12th European Conference on Object-Oriented Programming*, pages 186–211, July 1998.

[FDBC99] Adrian Friday, Nigel Davies, Gordon Blair, and Keith Cheverst. Developing adaptive applications: The most experience. *Journal of Integrated Computer-Aided Engineering*, 6(2):143–157, 1999.

[GH91] MG Gouda and T Herman. Adaptive programming. In *ieeetse*, volume 17, pages 911–921, 1991.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Designs Patterns: Elements Of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[Har98] Fazilah Haron. *Phase-based Adaptive Dynamic Load Balancing for Parallel Tree Computation*. PhD thesis, University of Leeds, 1998.

[HK00] Charles Herring and Simon Kaplan. The viable system architecture. In *Thirty-Fourth Hawaii International Conference on System Sciences (HICSS-34)*, 2000.

[HKC+99] Christopher K. Hess, Fabio Kon, Roy H. Campbell, Manuel Román, Dulcineia Carvalho, and Luiz Magalhães. Dynamic resource management for smart environments: The 2k approach. In *Inter-agency Workshop on Smart Environments*, Atlanta, Georgia, July 25-26 1999. Georgia Institute of Technology.

[HMS99] Ilija Hadzi'c, William S. Marcus, and Jonathan M. Smith. Policy and mechanism in adaptive protocols, 1999.

[KBE99] Mieczyslaw M. Kokar, Kenneth Baclawski, and Yonet A. Eracar. Control theory-based foundations of self-controlling software. *IEEE Intelligent Systems*, pages 37–45, May 1999.

[KC99] Fabio Kon and Roy H. Campbell. Supporting automatic configuration of component-based distributed systems. In *Proceedings of the 5th USENIX Conference on Object Oriented Technologies and Systems, COOTS'99*, may 1999.

[KP89] M.A. Krasnosel'skii and A.V. Pokrovskii. *Systems with hysteresis*. Springer, Berlin, 1989.

[Lad98] Dr. Robert Laddaga. Self-adaptive software. Technical report, DARPA, january 1998.

[Lad99] Robert Laddaga. Creating robust software through self-adaptation. *IEEE Intelligent Systems*, pages 26–29, May 1999.

[LL95] Karl Lieberherr and Cristina Lopes. Workshop on adaptable and adaptive software. Technical report, Northeastern University, 1995.

[MP90] H. Massalin and C. Pu. Fine-grain adaptive scheduling using feedback. *Computing Systems*, 3(1):139–173, winter 1990.

[OGT+99] P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, pages 54–62, May 1999.

[RKC99] Manuel Román, Fabio Kon, and Roy H. Campbell. Design and implementation of runtime reflection in communication middleware: the dynamictao case. In *Proceedings of the ICDCS'99 Workshop on Middleware*, June 1999.

[RSZ87] K. Ramamritham, J.A. Stankovic, and W. Zhao. Meta-level control in distributed real-time systems. In *7th International Conference on Distributed Computing Systems*, pages 10–17, september 1987.

[SA00] M.T. Segarra and F. André. A framework for dynamic adaptation in wireless environments. In *Proc. of TOOLS Europe 2000*, june 2000.

[SKK+90] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[SS98] Margo Seltzer and Christopher Small. Self-monitoring and self-adapting operating systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, 1998.