
Towards a Practical Methodology for Agent-Oriented Software Engineering with C++ and Java™

Michael J. Pont¹ and Emanuela Moreale²




BTSP: Software Engineering,
Department of Engineering,
University of Leicester,
University Road,
Leicester,
LE1 7RH

Leicester University, Department of Engineering,
Technical Report 96-33, December 1996.

Printing history:

First printed:	December, 1996
Re-printed (with corrections)	March, 1997.
Re-printed (with corrections)	April, 1997.

¹  M.J.Pont@sun.engg.leicester.ac.uk

²  em5@leicester.ac.uk

Abstract

In this report, we describe a practical methodology for agent-oriented software development. We begin by providing a working definition for the term 'agent' and a comparison of objects and agents as software building blocks. We then outline a straightforward analysis and design methodology for agent-oriented software development. For simple (single agent) systems, we go on to illustrate how an implementation can be effectively carried out using C++; for more complex (multi-agent) systems, we outline how such systems can be implemented in Java.

Acknowledgements

We thank all the members of 'BTSP: Software Engineering' for helpful comments on an earlier draft of this report.

Java is a trademark of Sun Microsystems, Inc.

1. Introduction

Over the last few years, we have seen a massive increase in new software engineering methodologies for object-oriented (O-O) software development (Booch, 1994; Coad and Yourdon, 1991; Graham, 1994; Jacobson *et al.*, 1993; Meyer, 1988; Rumbaugh *et al.*, 1991; Shlaer and Mellor, 1988). More recently, there has been increased interest in the use of software agents (Wooldridge and Jennings, 1995; Wooldridge *et al.*, 1996). There has, however, been relatively little published work describing methodologies for A-O software engineering (but see Kinny *et al.*, 1996).

In this report, we describe the foundations of an A-O software development method. The work builds on an integrated methodology for process-oriented, data-oriented, object-oriented and agent-oriented development which we have developed over the last four years (see Figure 1): this methodology is known as 'Integrated Software Engineering' (ISE). In the present report we describe the A-O component of ISE

It should be emphasised that ISE is not a new methodology: rather, it is a 'meta-methodology', or 'glue' methodology, intended to support the use, and integration, of a range of existing methodologies. For example, ISE can be used to ease the conversion of process-oriented designs (perhaps developed using 'Yourdon'), into object-oriented designs (perhaps using the 'Booch' or 'OMT' notations).

A complete description of a simplified version of ISE has been given elsewhere (Pont, 1996); a description of the latest version of ISE will be available shortly (Pont, in preparation).

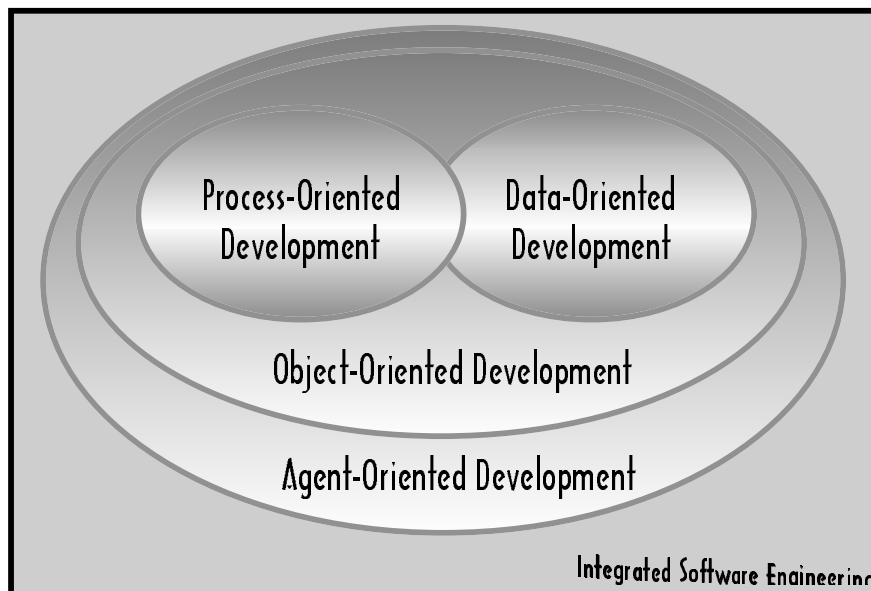


Figure 1: An overview of the relationship between the different methodologies included in the ISE toolbox. Note that data-oriented development encompasses both file-oriented development (with implementation typically in COBOL), and entity-oriented development (with implementation typically in SQL).

2. Why we need agents

Agent-oriented (A-O) software engineering represents a novel and, we believe, powerful way of approaching certain large-scale, complex, software engineering problems.

The A-O approach we will describe in this report builds on well-established O-O techniques. However, as we will attempt to demonstrate, A-O and O-O approaches differ in a number of important respects. We therefore begin by comparing an O-O and A-O approach to the development of software simulations, and then go on to consider the application of A-O techniques to more general software problems.

2.1 Simulations

One area in which O-O techniques have been applied is in the development of simulations of real-world processes. This process is significant in part because one of Stroustrup's inspirations in creating C++ was the programming language Simula, a language originally developed to allow the creation of such simulations. Stroustrup (1991) himself outlines the use of C++ for modelling traffic flow in a city, suggesting that such a simulation might be used to determine the best place to locate the fire stations in order to minimise the travel time to a fire at any period of the day or night.

Suppose that we wished to develop a simulation such as that proposed by Stroustrup. We might start by thinking about one important class of vehicles, the car (Figure 2).

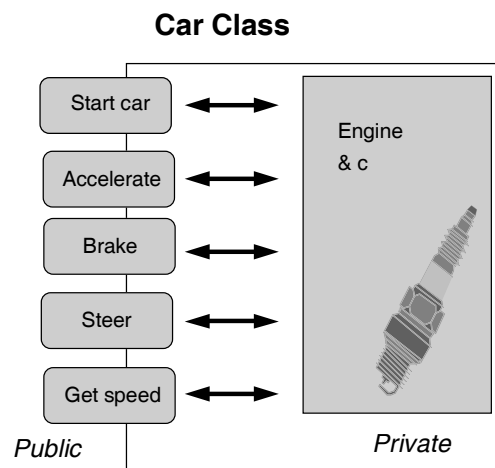


Figure 2: A schematic representation of a “car class”.

This car model emphasises that, as far as the average user is concerned, the various methods - “Start car”, “Accelerate”, “Brake”, and so on - are all they are aware of (Figure 3). They do not wish to know anything about the engine or other mechanics (at least until something goes wrong...). These methods therefore form a useful interface, separating the user from the underlying complexity of the engine, electronics and mechanics that power and control the car. This interface allows the user to communicate with and control the underlying “data” and “processes” in a useful way.

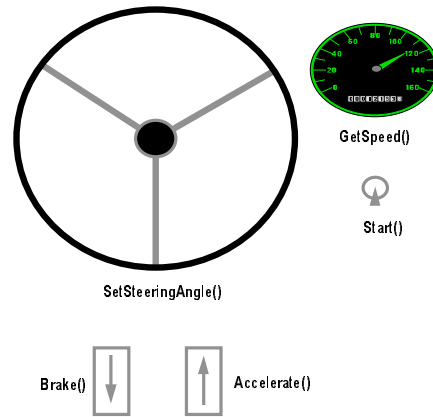


Figure 3: Components of the interface to a simple car class, with labels suggesting how we might translate the interface components into C++ member functions.

We can also see how we might go about constructing such a simulation in C++. For example, a simple car class compatible with Figure 2 could be defined as follows:

```
class cCar
{
public:
    void Start();
    void Accelerate();
    void Brake();
    void Set_Steering_Angle(const int);
    int GetSpeed();

private:
    int _Speed;
    int _Steering;
};
```

Following such a class definition (and the creation of appropriate member function definitions), we could create our car object with the following line:

```
cCar Volvo480; // Create a Volvo car
```

We could then ‘start’ this car, and simulate it with a few lines as follows:

```
cCar Volvo480; // Create a Volvo car
Volvo480.Start(); // Start the car
Volvo480.Accelerate(); // Drive away
```

We could attempt to complete this simulation, using an object-oriented approach, and could certainly achieve this. However, we suggest, this may not be the best way to think about this type of software.

One way of viewing this problem is that we have successfully modelled a car: however, what we were really interested in modelling was the car, *plus the car’s driver*. If we view the car, plus driver, as an *agent*, then we might consider the methods and data required to simulate the car: we would also, crucially, however think of the aims of this simulated object. In our traffic simulation, for example, the driver of our Volvo car might have the aim of leaving work at five o’clock and driving home (unless the weather is fine, in which case he may leave early and go to play golf). This aim will have a very significant impact on our simulation. Clearly, this is an

important part of the car we are simulating: it is not some global controlling factor. Because this aim is ‘part of’ the car agent, we should include it in the car agent.

In an O-O program, as we illustrated above, our car object has no autonomy: it is told what to do (in a C++ program, for example, this may be through calls to member functions in `main()`). By contrast, in an A-O program, the agents will each have aims, which they follow, in parallel, without being given instructions from ‘outside’. The aims represent the agent’s ‘own agenda’: that is, what it will attempt to do when not being given other instructions. The aims are kept distinct from the services that the agent can provide (Figure 4).

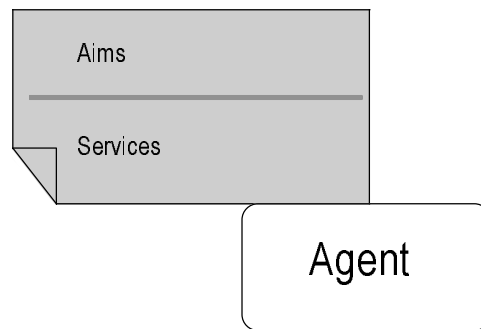


Figure 4: *A general annotated agent.*

For example, returning to our traffic simulation, we can split the aims and services of one particular car agent as shown in Figure 5.

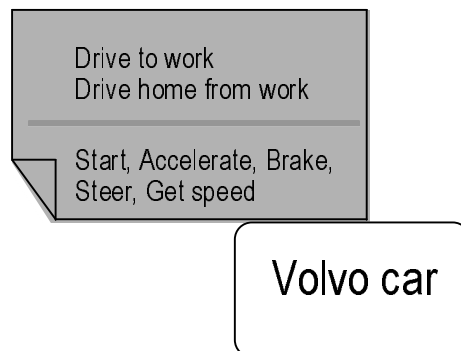


Figure 5: *The annotated car agent from the traffic simulation example, showing the agent aims (‘Drive to work’, ‘Drive home from work’) and services (‘Start’, ‘Accelerate’, ‘Steer’, ‘Get speed’).*

We will consider how such systems can be implemented, in C++ and Java, in Section 6.4.

2.2 More general use of A-O techniques

We believe that the arguments, outlined above, that simulations of animate objects are better viewed as collections of agents than as collections of objects is a compelling one. However, as we will argue here, we also believe that for many, more general, software development techniques, an A-O approach can also be effective.

To consider why this is the case, we need to go back to the 1970s. At this time, as structured methods were developing, we would view users of a computer system as ‘data sources’ and ‘data sinks’ (Yourdon, 1989). For example, suppose we wished to create the software required to control an auto teller machine (ATM) with the interface shown in Figure 6.

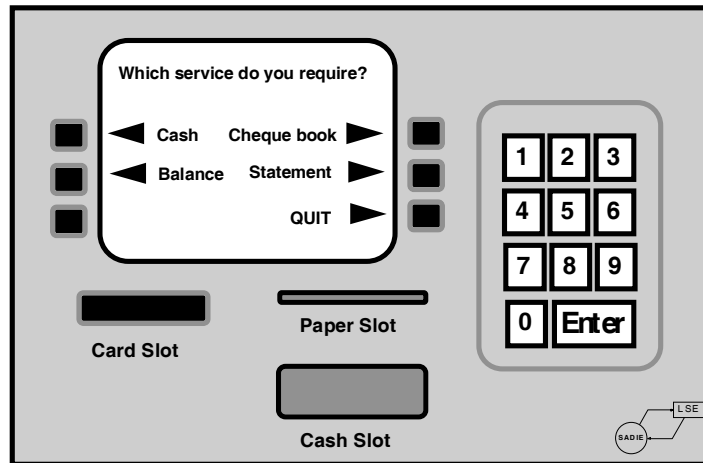


Figure 6: The interface to the ATM system. From Pont (1996).

The system will be used by customers of the bank mainly to collect cash (Figure 7).

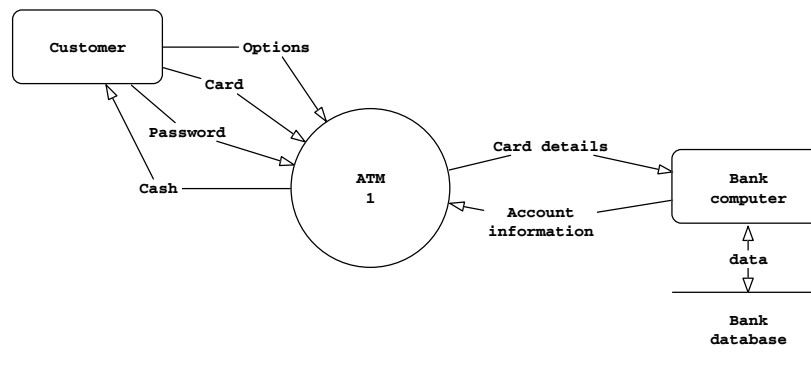


Figure 7: Partial data domain model (context diagram) for a simple (F-O) ATM system.

The problem with this approach is that, in a sense, we are attempting to represent the world as a ‘function library’ and users as ‘sources of data’. This is the ‘tail wagging the dog’: we should not be adjusting our world view to match our implementation technique: instead, we should change the way we implement - and think about software - so that it matches the way the world actually is.

This approach was, of course, the driving force for an approach to software development in the 1980s whereby complex systems are viewed at a higher level as collections of objects, interacting through a process of message passing (Figure 8). The idea of objects is attractive, and perhaps the single largest advantage of an O-O approach is that ‘models’ of the system, developed by the analyst and designer, appear natural to those for whom the system is being developed.

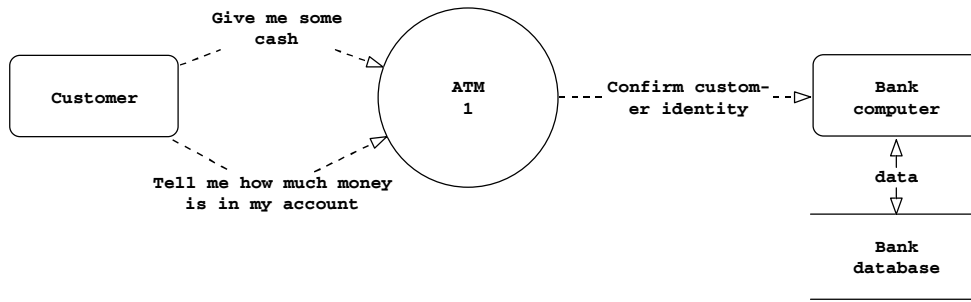


Figure 8: Partial message domain model for a simple (O-O) ATM.

However, we can now take this approach one small but important step further. The objects in an O-O model are passive in nature, yet objects in the real world do not simply wait around for your system to ‘invoke a method’: they have their own aims, and plans. We define such active, concurrent objects as agents. Using such agents, we continue to model the world as a group of communicating individuals and systems. The idea of message passing is maintained, but we add to this agent *aims* (Figure 9).

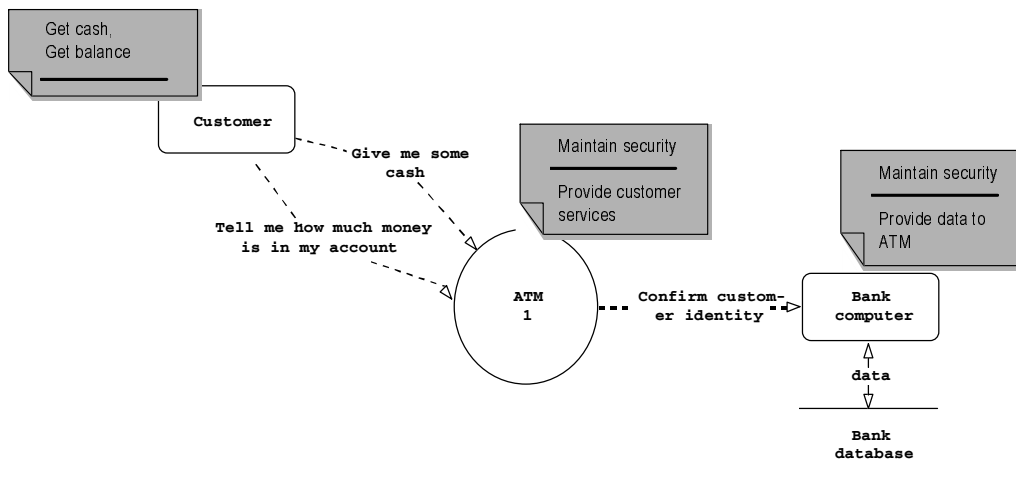


Figure 9: Partial message domain model for a simple (A-O) ATM. We will consider the development of such diagrams in Section 4.3.

The concept of ‘active objects’ makes sense, we believe, for a range of applications. Within a business context, for example, aims are an ideal way of representing the roles of different individuals within an organisation, and for representing business rules. Alternatively, when developing applications to integrate with complex desktop computer software, the aims may cover background tasks, such as spell checking or reformatting. Similarly, at a system level, the aims of agents may encompass much lower-level concepts such as memory ‘garbage collection’.

As we will see during the course of the remainder of this report, an A-O philosophy involves representing the software system we are developing, and the other systems (and people) it interacts with as ‘animate objects’. We perform the process of software development by anthropomorphising: that is, by viewing our software components as *human*. We develop the system by considering the agents making up the system, recording their aims and viewing the ‘message passing’ between agents as a form of conversation. In short, by viewing our

software system as an agent, we are saying that the behaviour of this system is sufficiently complex that it may be best viewed as being alive.

3. An overview of A-O analysis

Throughout the remainder of this report we describe a simple practical methodology for A-O analysis and design, and provide suggestions for implementations in C++ and Java. In the methodology we introduce here, we maintain a traditional and, we believe, valuable distinction between the analysis phase of the project and the design phase. As one would expect, we focus primarily on logical issues during analysis: that is on what the software is to do, rather than how this result is to be achieved. During design, we attempt to translate these logical requirements into a physical model: as far as is feasible, we endeavour to create a physical model which is implementation independent. As we will see in Section 6.4, however, this last aim is not always easy to achieve as few computer languages (yet) provide much support for the implementation of multi-agent systems.

We outline the four-stage analysis process used in ISE for A-O development below. Following this brief overview, we provide an example of the technique in practice.

A 1. The external agents

The external agents are identified, and their aims are highlighted.

A 2. The logical interaction ('conversation') diagrams

An interaction diagram is created, embodying a 'walkthrough'.

A 3. The system and external messages

The messages to which the system must respond are identified.

A 4. The message domain diagram

The message domain diagram is produced.

Note that, in this report, we have focused on the core technical issues relating to the implementation and not discussed business issues, such as software size estimation or management of A-O projects. These issues are discussed in detail elsewhere (Pont, in preparation).

Note also that this A-O methodology is compatible with the techniques for process-, data- and object-oriented development described elsewhere (Pont, 1996)

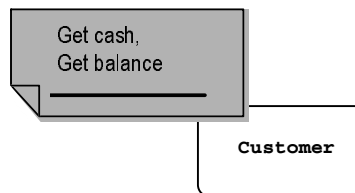
4. A practical example of A-O analysis

As an example, we will consider the ATM introduced in Section 2.2 above, and outline how such software might be developed using an A-O approach. To complete the analysis of this system, we will follow the four-stage process outlined above.

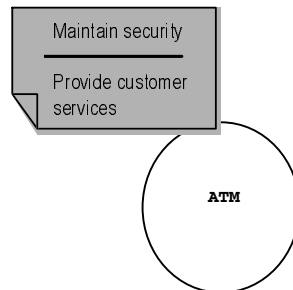
4.1 The external agents

As with any form of systems analysis, we begin at the outside of the system, and work our way in. We begin by considering who will use the system, what other systems (e.g. databases) our system must interact with, what hardware our system will interact with.

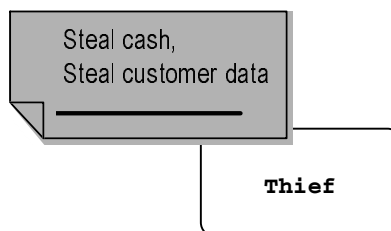
Users will, generally, have a list of aims which they wish the system to satisfy: in the case of our simple ATM system, we will assume that the customers simply wish to obtain cash and to find out the balance of funds in their account:



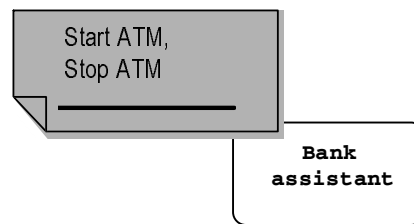
By contrast, we view the overwhelming aim of the ATM system is to maintain its security and to protect itself from attack:



As we consider this security issue in more detail, we might usefully add an extra 'customer' to the MDD to highlight this: a 'thief', who has the aims of stealing money from the ATM, and / or gaining access to the central bank computer and stealing account details:



As we develop this system, we need to consider who activates the system in the first place: we'll assume it is an assistant at the bank:



4.2 The logical interaction diagrams

During the analysis and design process, a representation of the data structures at the heart of the software system is created, in the form of an agent diagram and a class-relationship diagram. Together, these diagrams provide information on two aspects of the system in which we are interested:

- **Process:** what does it do?
- **Data:** what does it do it to?

However, this information alone is insufficient to fully specify the system: we need to be able to add a further feature to this list:

- **Control:** when does it do it?

Any software system, whether process-, data-, object- or agent-oriented, needs to include these three perspectives: process, data and control.

In an agent-oriented system, interaction diagrams (Booch, 1994) provide a natural way of representing control and timing information. Typically the provisional interaction diagrams arise as the result of a walkthrough: walkthroughs are a widespread and cost-effective way of reviewing and refining the system requirements (Norris *et al.*, 1993).

When developing with agents, we view the world as being composed of 'intelligent things'. We begin to model the interaction between these things in the form of a conversation. For example, suppose we wish to develop a simple control system for a greenhouse system. We can begin to think about the way the system will operate by considering the 'conversation' between the control system we are constructing, and the system externals (a temperature sensor, 'cooler', and 'heater'), as illustrated in Figure 10.

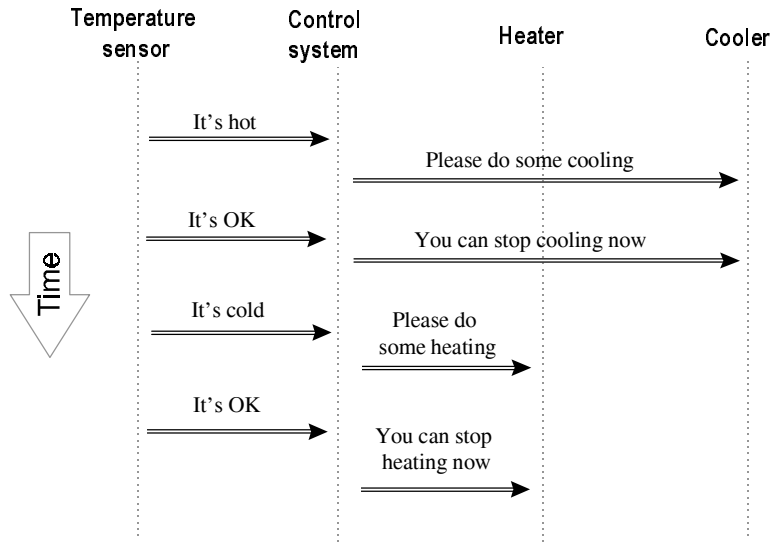


Figure 10: A 'conversation' between agents in a simple greenhouse control system.

Typically, for a more realistic system, an interaction diagrams will be drawn up during the course of a walkthrough, as the operation of the system is considered over a typical period of use: As an example of the end result of this process, Figure 11 shows how the ATM system might operate while dispensing cash to a customer.

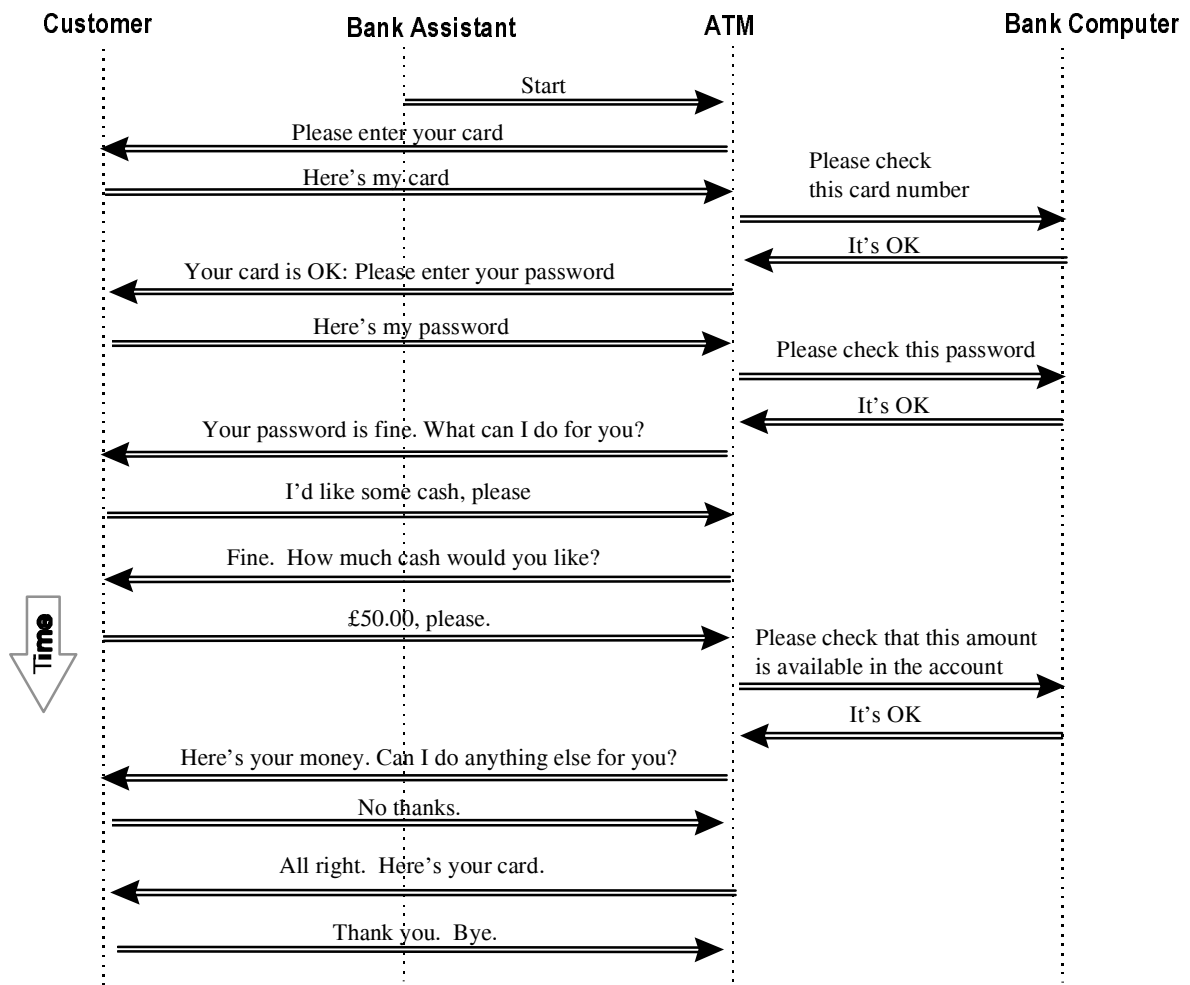


Figure 11: A logical interaction diagram. It can help to view these as conversations between the various agents in the system.

4.3 The system and external messages

The next phase in the analysis process involves identifying the key system and external messages: that is, the messages to which the system must respond, and (where appropriate) to which any associated externals must respond.

Examining the various logical interaction diagrams often provides a powerful way of assisting in the identification of important messages. For example, Figure 11 reproduces Figure 12, this time with some possible system messages identified. This process would be repeated for any remaining logical interaction diagrams.

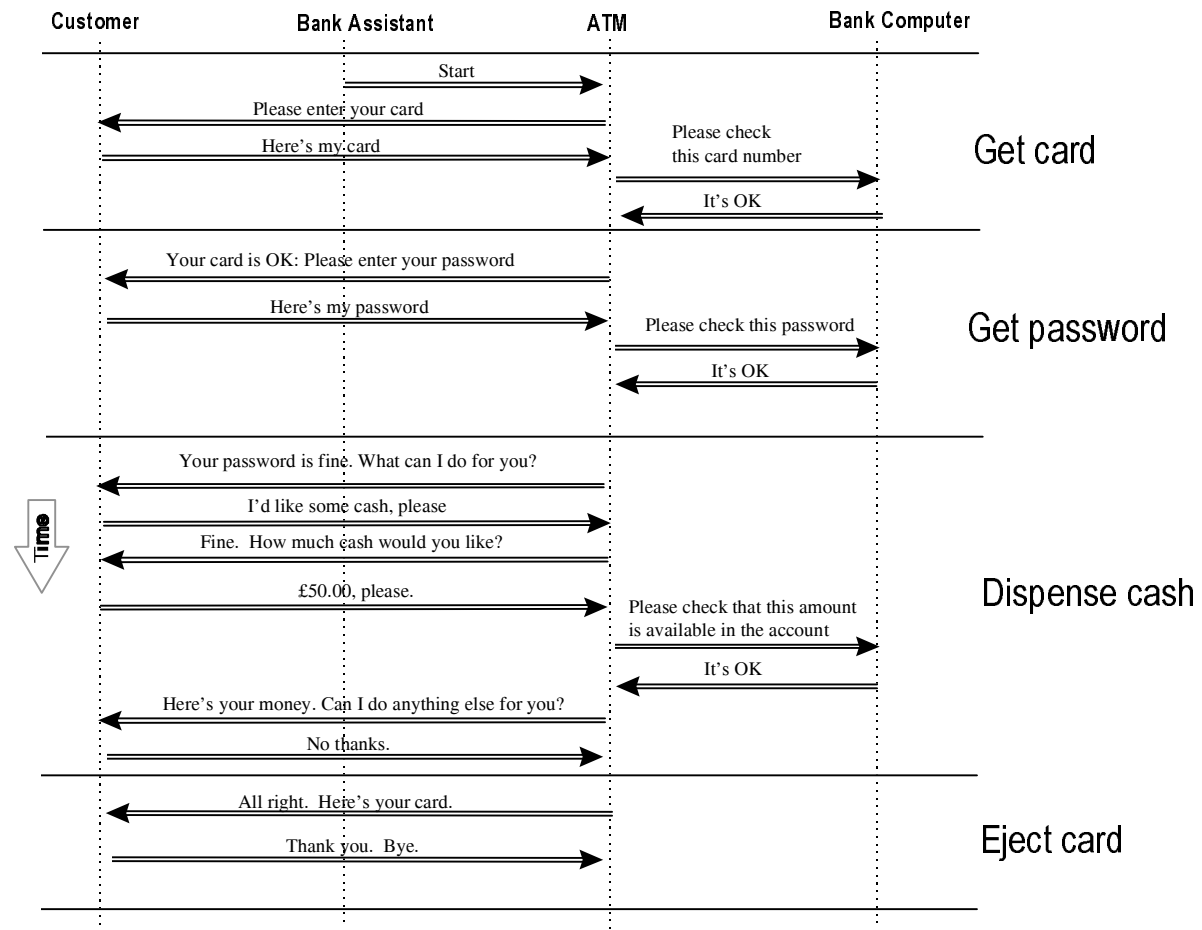


Figure 12: An annotated logical interaction diagram.

4.4 The message domain diagram

The final analysis stage involves the production of the message domain diagram (MDD). As we create the MDD, we are completing our summary of what the system must do, from the perspective of the people and other agents with which it must interact. We have already shown a possible, provisional, message domain diagram for the ATM system (Figure 9).

The finished message domain diagram is shown in Figure 13.

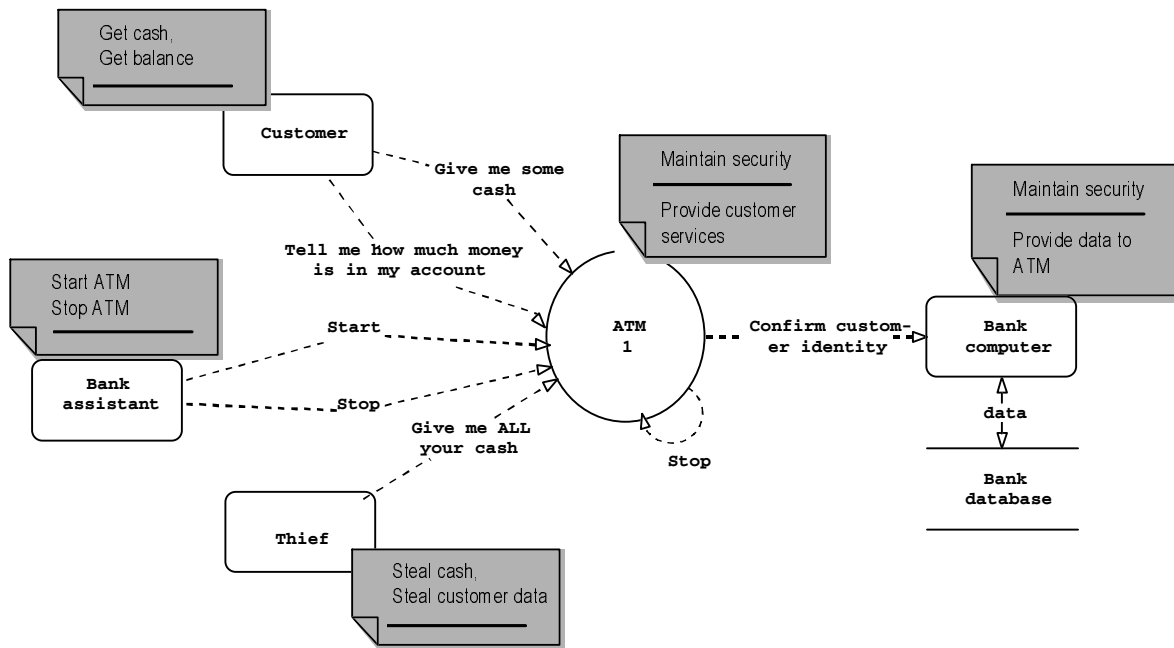


Figure 13: Message domain diagram for the simple (A-O) ATM.

5. An overview of A-O design

As in the analysis phase, we present first an overview of the A-O design process, then provide a worked example to illustrate key components of this process.

D 1. The internal agent diagram

An internal agent diagram is produced. The aims of the agents, and the services they provide is considered.

D 2. The class-relationship diagram

The agent diagram is translated into a class-relationship diagram (CRD) by adding in the relationships.

D 3. The physical interaction diagrams

Physical interaction diagrams are created for each main process, based on the logical interaction diagrams described above.

It should be emphasised that the various diagrams and specifications developed during the analysis and design phases are interrelated as illustrated in Figure 14.

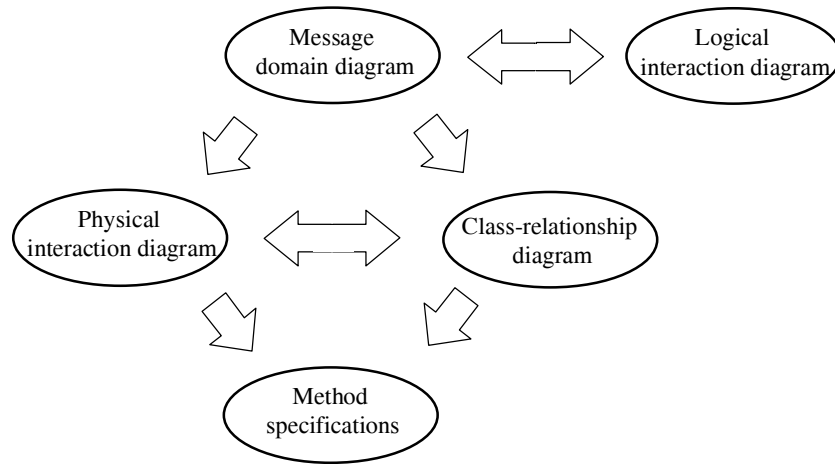


Figure 14: *The relationship between the various diagrams and specifications used in the A-O design process.*

6. A practical example of A-O design

By way of example, we will continue the development of the ATM system begun in Section 4.

6.1 The agent diagram

The most basic technique for uncovering candidate data structures involves looking for nouns in the initial problem statement (Abbott, 1983). This process is essentially identical whether we are looking for entities, objects (Pont, 1996) or agents: we will therefore not pursue this further here.

One possible agent diagram resulting from such an analysis is given in Figure 15.

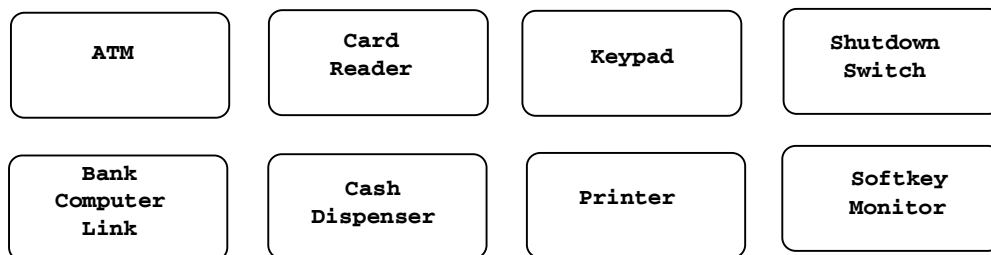


Figure 15: *The agent diagram. Note that we are concerned in this diagram only with internal agents: the externals, identified in Figure 13, don't appear again here.*

Next we consider the agents identified in the previous step, and try to clarify what each aims to do, and the services it provides. At this stage, no attempt is made to view either aims or services in terms of their eventual implementation, but simply to record these by appending 'notes' to our provisional agent diagram. As we do this, we begin to refine our description of the agents in the system. This process begins as the original agent diagram is created and continues through the analysis and design phases.

The end result of this process is the annotated agent diagram, like that shown in Figure 15.

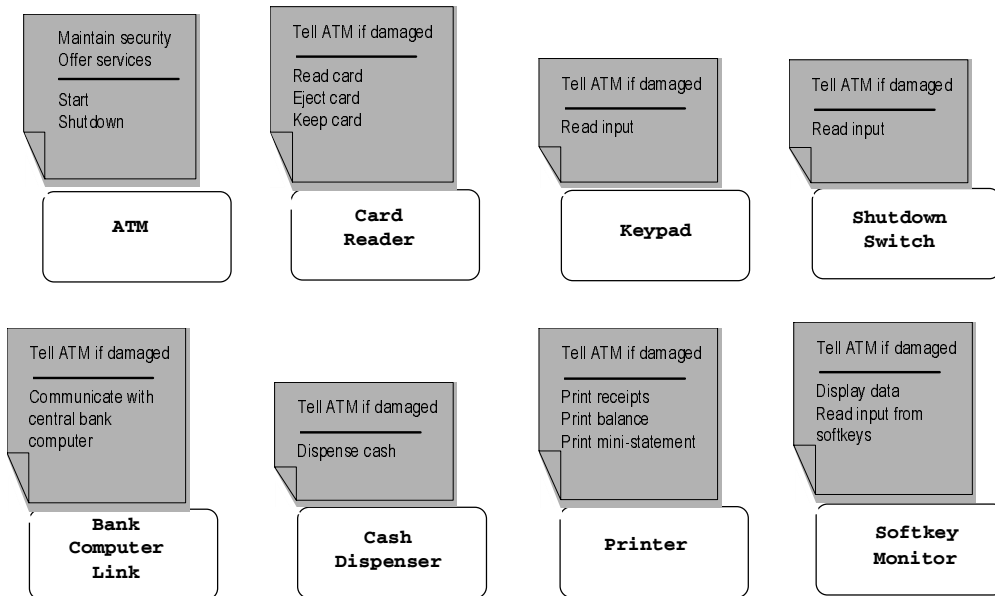


Figure 16: The annotated agent diagram.

6.2 The class-relationship diagram

The first thing to emphasise is that agents, like objects, are instances of classes: thus, a class-relationship diagram, widely used in O-O methodologies, is also an essential component of our A-O methodology.

An example of such a CRD is shown in Figure 17.

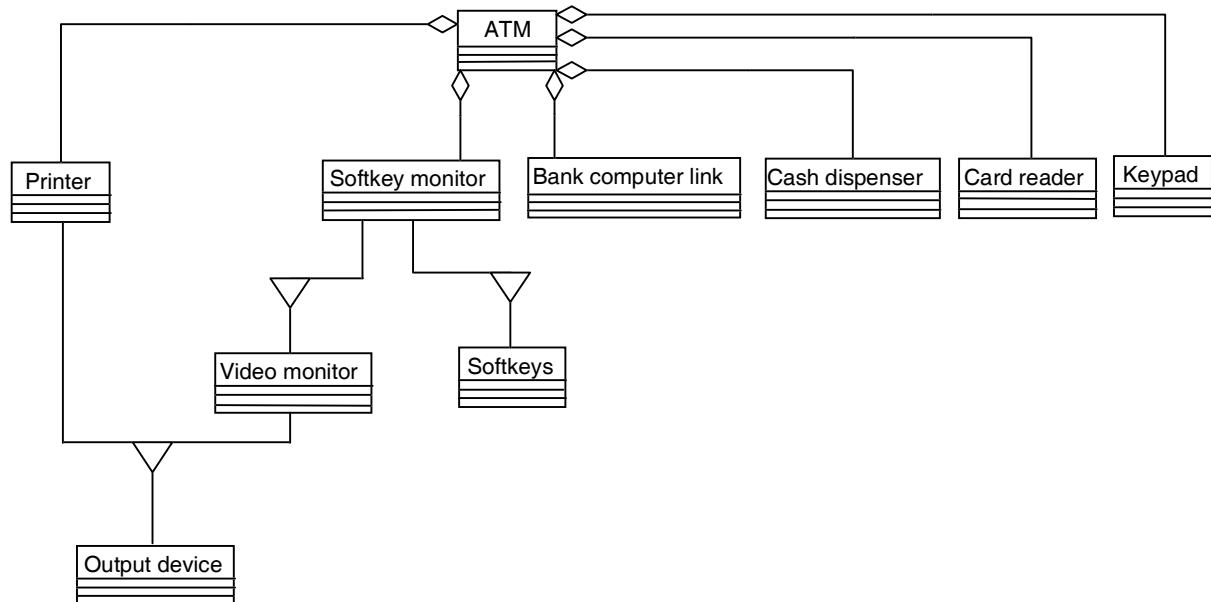


Figure 17: The ATM class-relationship diagram (CRD).

6.3 The physical interaction diagrams

In this stage, the logical interaction diagrams are refined, and evolved into physical interaction diagrams. In the process, the ‘notes’ concerning the flow of messages between different agent in the system, are translated into a series of invoked methods (that is, a series of member function calls).

An example of the result of this stage is given in Figure 18.

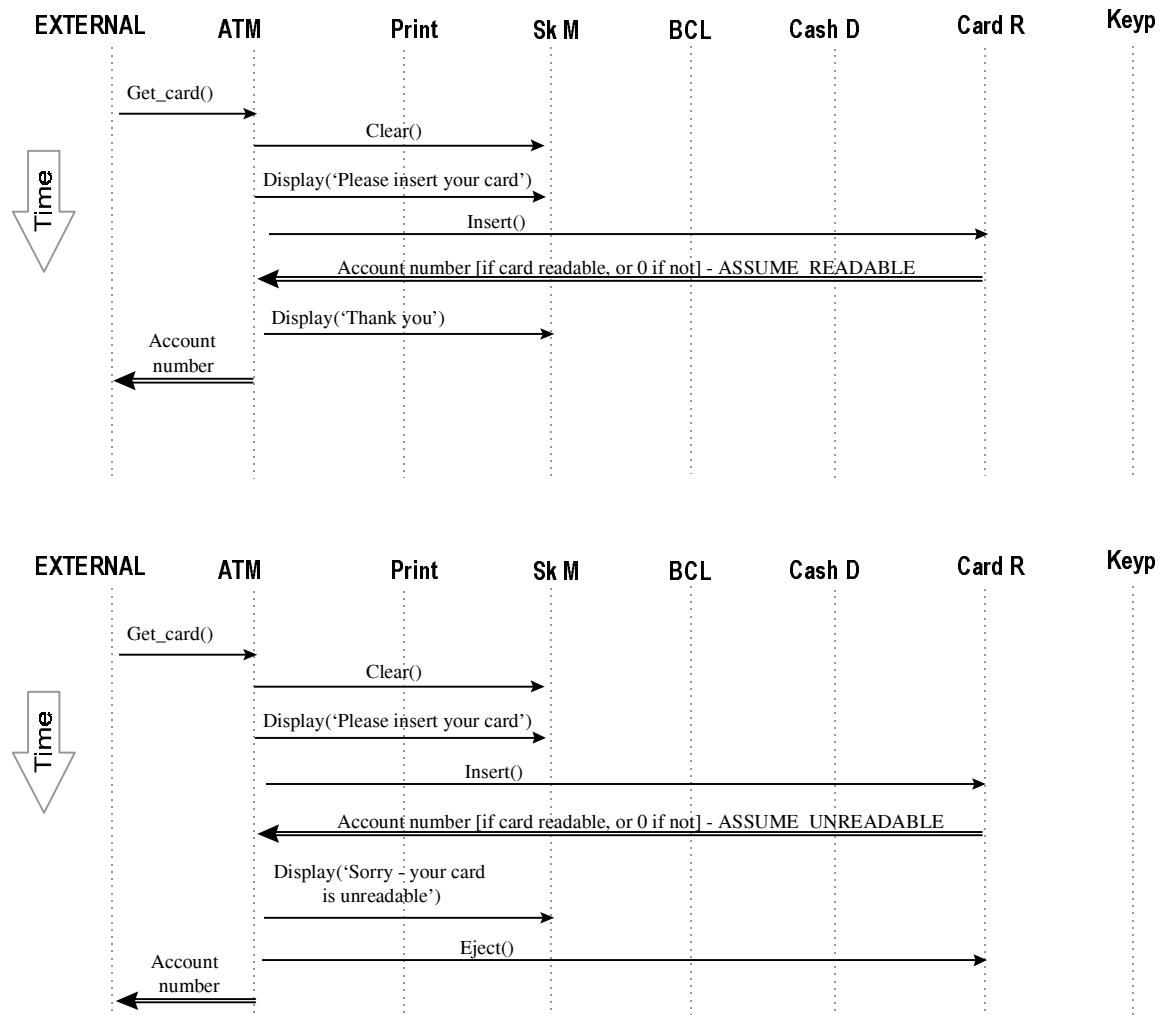


Figure 18: Physical interaction diagrams for the ‘Get_card()’ process. The upper diagram illustrates the operation when a readable card is inserted: the lower diagram illustrates the operation when a damaged card is inserted.

6.4 The method specifications

We begin this step by considering the application’s autonomy function, then consider all other member functions (methods). We can specify each type of function in an ‘implementation independent’ way by using notes, structure English or pseudo-code. Alternatively, they can be simply written in the form of code fragments.

We'll use code fragments here, to illustrate possible implementations in C++ and Java. For example, consider the main autonomy function for the `cATM` class. This might be implemented, in C++, as shown in Code fragment 1.

```

void cATM::Start()
{
    int acc;
    bool Valid_card_entered;
    bool Valid_password_entered;

    try
    {
        // Treat all shutdown requests as 'interrupts'
        for(;;)
        {
            Prepare_For_Next_Customer();

            Valid_card_entered = Get_Card();

            if (Valid_card_entered)
            {
                Valid_password_entered = Get_Password();

                if (Valid_password_entered)
                {
                    // Keep delivering services till user
                    // says quit (or hardware fault)
                    Deliver_Services();

                    Eject_card();
                }
            }
        }
    }
    catch(...)
    {
        Shutdown();
    }
}

```

Code fragment 1. The prototype A-O ATM system (`Start()` member function only). C++ code.

If we implement in C++, then the agent-oriented version of this system might have a `main()` function as shown in Code fragment 2.

```

int main()
{
    cATM cash_machine;

    cash_machine.Start()
}

```

Code fragment 2. The prototype A-O ATM system (function `main()` only). C++ code.

What we have done in code fragments 1 and 2 is to implement the ATM as a single-agent system. There is an important conceptual difference between a 'pure' object-oriented system, and a system containing a mixture of objects plus a single agent. However, at a practical level, this distinction is rather subtle and comparatively unimportant.

When agent-oriented systems become more interesting, and more powerful, is when more than one agent is involved. Suppose, for example, we wished to implement the ATM system as a

multi-agent system. As outlined during the analysis and design phases above, we would then consider each of the components of the ATM to be an agent, with the 'aim' of 'constantly' checking its own hardware. The ATM components will continue to perform this operation, unless instructed to do something else by the ATM agent itself.

Clearly, such an implementation requires support for concurrency (multi-threading): we cannot directly support such a requirement in C++, but we can do so in Java, as shown in Code fragment 3.

```

public class cATM
{
    public static void main(String args[])
    {
        cATM cash_machine = new cATM();

        cash_machine.Start();
    }

    ...

    private cKeypad          _Kp = new cKeypad();
    private cSoftKeyMonitor  _SKM = new cSoftKeyMonitor();
    ...
    private cPrinter         _Pr = new cPrinter();

}
////////////////////////////////////

public void Start()
{
    int acc = 0;
    boolean Valid_card_entered = true;
    int tries;
    boolean Valid_password_entered = false;
    boolean user_wants_to_quit = false;

    System.err.println("*** Starting CD thread... ***");
    _CD.start();

    System.err.println("*** Starting BCL thread... ***");
    _BCL.start();

    ...

    try
    {
        // Treat all shutdown requests as 'interrupts'
        for(;;)
        {
            Prepare_for_next_customer();

            // acc == 0 means card not valid
            acc = Get_Card();

            if (acc != 0)
            {
                tries = 0;
                user_wants_to_quit = false;

                tries++;
                Valid_password_entered = Get_password(tries, acc);
            }
        }
    }
}

```

```

        if (Valid_password_entered)
        {
            // Keep delivering services till user
            // says quit (or hardware fault)
            Deliver_services();
            Eject_Card();
        }
    }
}

catch(Exception e)
{
    Shutdown();
}

return;
}

```

Code fragment 3. The prototype A-O ATM system (fragment). Java code.

Note the similarity in the code created for both the C++ and Java implementations.

7. Future work

It is clear from our outline here that many of the basic tools and techniques required to support A-O development are already in use by O-O methodologists. However, there are two remaining areas where A-O development differs significantly from O-O development which we have not addressed in this report and which, we suggest, require further research.

The first area is that of collaboration and co-operation between agents. For example, we can imagine a scenario where certain aims are those of a group (e.g. workers in the same company) rather than being restricted to a single individual. Further work is required to consider how such 'group aims' should be recorded and implemented.

Secondly, a major area of concern is in the testing of A-O systems. We have previously highlighted (Pont, 1996) some of the complexities of testing O-O systems. A-O systems share all of these complexities, and also introduce further potential problems resulting from the concurrent nature of the implementation. This is an area which must be addressed if A-O techniques are to be used in practical systems.

8. Conclusions

In this paper, we have provided a simple definition for agents as 'objects with aims'. We have attempted to highlight the distinction between O-O and A-O system development, and to argue for some of the benefits of adopting an A-O approach to large-scale software development. Finally, we have outlined a methodology for agent-oriented (A-O) software engineering, and suggested some areas for future work in this area.

A-O software development techniques are still in their infancy: however, as we have tried to demonstrate in this paper, the use of agents provides a simple, logical and powerful extension to well-tried O-O approaches to software development.

9. References

- Abbott, R.J.** (1983) "Program design by informal English descriptions," *Comm. ACM*, **26** (11): 882-894.
- Booch, G.** (1994) "*Object-Oriented Analysis and Design*," Benjamin Cummings.
- Coad, P. and Yourdon, E.** (1991) "Object-Oriented Analysis" (2nd Ed.), Yourdon Press.
- Kinny, D., Georgeff, M.P. and Rao, A.S.** (1996) "A methodology and modelling technique for systems of agents," in van de Velde, W. and Perram, J.W. [Eds.] 'Agents Breaking Away - 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World [MAAMAW '96]'. Springer-Verlag [Lecture Notes in Artificial Intelligence, Volume 1038]. Pp. 56-71.
- Graham, I.** (1994) "*Object-Oriented Methods*," (2nd Ed.) Addison-Wesley.
- Jacobson, I., Christerson, M., Jonsson, P. and Övergaard, G.** (1993) "*Object-Oriented Software Engineering*," (Revised printing). Addison-Wesley.
- Macguire, S.** (1994) "*Debugging the Development Process*," Microsoft Press.
- Meyer, B.** (1988) "*Object-Oriented Software Construction*," Prentice Hall.
- Norris, M., Rigby, P. and Payne, M.** (1993) "*The Healthy Software Project: A Guide to Successful development and management*," John Wiley.
- Pont, M.J.** (1996) "*Software Engineering with C++ and CASE Tools*," Addison-Wesley.
- Pont, M.J.** (in preparation) "*Software Engineering: An Integrated Approach*," Addison-Wesley. Due for publication January, 1998.
- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorenzen, W.** (1991) "*Object-Oriented Modeling and Design*." Prentice-Hall.
- Shlaer, S. and Mellor, S.J.** (1988) "*Object-Oriented Systems Analysis: Modelling the World in Data*," Prentice-Hall.
- Stroustrup, B.** (1991) "*The C++ Programming Language*," (2nd Ed.) Addison-Wesley.
- Wooldridge, M. and Jennings, N.R. [Eds.]** (1995) "*Intelligent Agents - Theories, Architectures and Languages*," Springer-Verlag [Lecture Notes in Artificial Intelligence, Volume 890].
- Wooldridge, M., Mueller, J.P. and Tambe, M. [Eds.]** (1996) "*Intelligent Agents, Volume II*", Springer Verlag. [Lecture Notes in Artificial Intelligence, Volume 1037].
- Yourdon, E.N.** (1989) "*Modern Structured Analysis*," Prentice-Hall.

Table of Contents

1. Introduction	2
2. Why we need agents	3
2.1 Simulations	3
2.2 More general use of A-O techniques	5
3. An overview of A-O analysis	8
4. A practical example of A-O analysis	9
4.1 The external agents	9
4.2 The logical interaction diagrams	10
4.3 The system and external messages	12
4.4 The message domain diagram	12
5. An overview of A-O design	13
6. A practical example of A-O design	14
6.1 The agent diagram	14
6.2 The class-relationship diagram	15
6.3 The physical interaction diagrams	16
6.4 The method specifications	16
7. Future work	19
8. Conclusions	19
9. References	20