Development Methodologies

A methodology is a system of methods and principles used in a particular sub-discipline of software design. There are a large number of these, reflecting the way in which software design in practice has specialised. Those which are mature usually are supported by specialist tools and techniques.

A traditional view of design in software engineering is analogous to building a cathedral: we make careful, comprehensive blueprints; use these as the reference point for coordinating architects and craftspeople; and ensure that the individual components connect precisely in the way described in our blueprints. We shall look at an example of this style of design: the Unified Process (Section 3.1). This, however, is not the only approach available. We also can adopt a less prescriptive, opportunistic style of design where a greater emphasis is placed on rapid revision of blueprints and the artifacts built from them. The practices of Extreme Programming (Section 3.2) are an example of this.

3.1 The Unified Process

The Unified Process is a traditional "cathedral" style of incremental design driven by constructing views of a system architecture. It has the following key features:

- It is component based, commonly being used to coordinate object oriented programming projects.
- It uses UML - a diagrammatic notation for object oriented design - for all blueprints.
- The design process is anchored, and driven by, use-cases which help keep sight of the anticipated behaviours of the system.
- It is architecture centric.
- Design is iterative and incremental - via a prescribed sequence of design phases within a cyclic process.

3.1.1 Phases of Design Cycles

Design in the Unified Process proceeds through a series of cycles, each of which has the following phases:

Inception : produces a commitment to go ahead. By the end of this phase a business case should have been made; feasibility of the project assessed; and the scope of the design should be known.

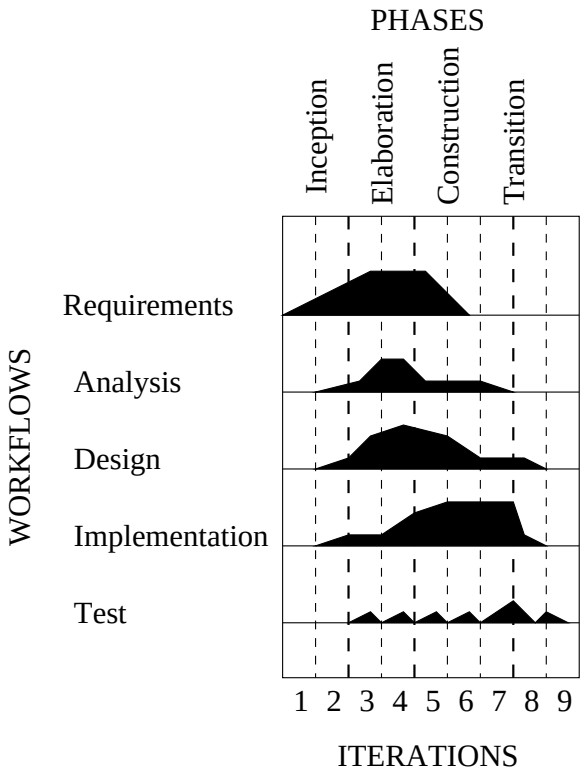

Figure 3.1: Phases in a cycle of the Unified Process

Elaboration : takes us to a working specification of the system. By the end of this phase a basic architecture should have been produced; a plan of construction agreed; all significant risks identified; and those risks considered to be major should have been addressed.

Construction : produces beta-release system. By the end of this phase a working system should be available, sufficient for preliminary testing under realistic conditions.

Transition : introduces the system to its intended users.

Within these phases we may go through a number of iterations, each involving the normal forms of workflow activity (requirements specification, analysis, design, implementation and testing). An example is shown in Figure 3.1. Here we see a series of nine iterations covering the four phases described above. The pattern of workflow across iterations is shown by the graph for each activity (the volume beneath each graph giving the amount of effort) - for instance we can see that requirements specification ramps up during the inception phase; stays high during the elaboration phase; then drops during the construction phase, becoming negligible by the time we get to the transition phase.

A principal product of the Unified Process is a series of models, each appropriate to a key stage in system design. Since many different models are produced, each for a different design purpose but all related to the same system, we need

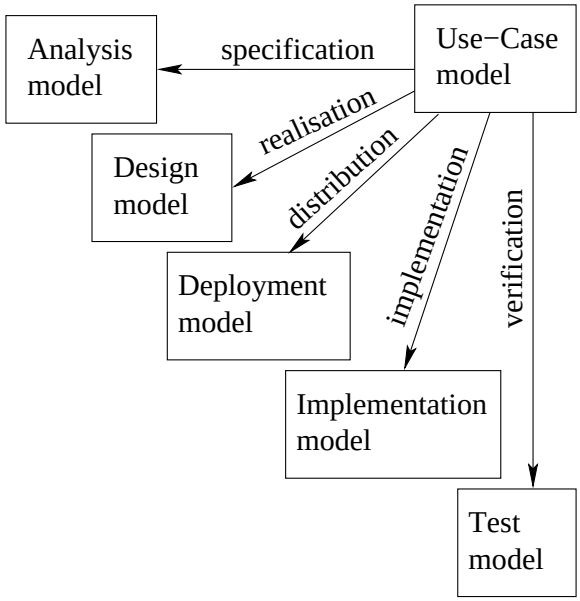

Figure 3.2: A series of models in the Unified Process

some common point of anchorage. This is provided by a use case model. Figure 3.2 shows a typical arrangement, in which five models appropriate to specific design activities all are rooted in the same use case model.

The purpose of a use case is to describe the functionality required of the system from the point of view of those concerned with its operation. The way a use case does this is by specifying a sequence of actions, including variants, that the system can perform and that yield an observable result of value to some actor. In the Unified Process, this drives:

- Requirements capture.
- Analysis and design of how system realises use cases.
- Acceptance/system testing.
- Planning of development tasks.
- Traceability of design decisions back to use cases.

In the next section we demonstrate this by example.

3.1.2 Example

In this example we will consider part of the design for a cash dispenser system. We begin with the simple, initial use-case diagram of Figure 3.3. The actor here is a customer and the actions are withdrawing money, depositing money and transferring between accounts.

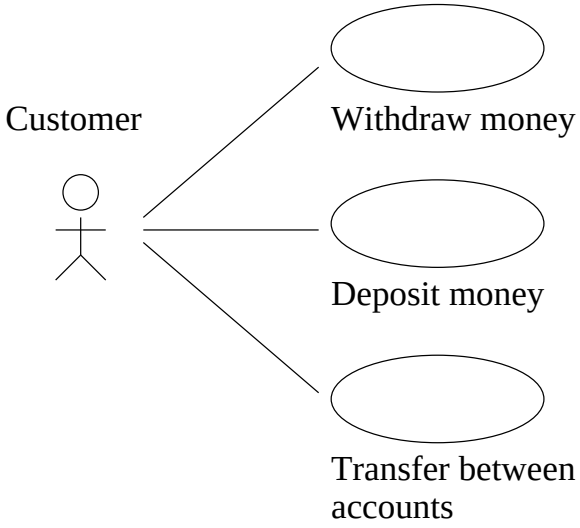

Figure 3.3: Initial use case diagram

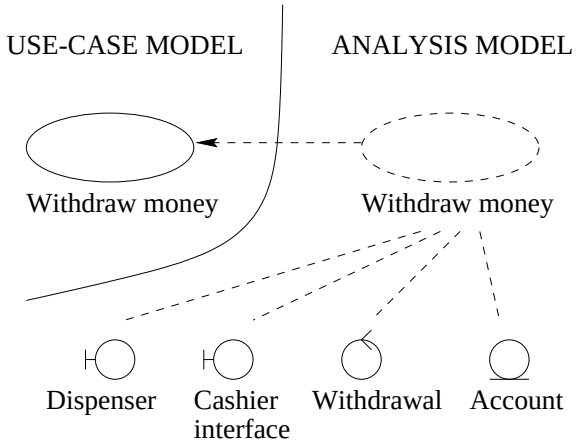

Figure 3.4: Analysis model for withdrawing money

Henceforth we shall concentrate on withdrawing money. Which are the main top-level elements (classes in object oriented design) necessary in our system to perform this action. To depict this we produce an analysis model as shown in Figure 3.4. This shows four analysis classes: a dispenser interface; a cashier interface; a process for making withdrawals; and an account database. Notice how interfaces, processes and databases are distinguished using different symbols, which will recur in subsequent diagrams. Also notice that we link the analysis model to the use case model so as to have traceability back to our use case.

Now we have our analysis classes we show how they interact, with each other and with the customer (the actor in our use case model). Figure 3.5 shows this collaboration model. Customer interaction is with the cashier interface (identifying the customer) and the dispenser (passing out money). Both of these interact with the withdrawal process which in turn interacts with the account database.

The next question is: what object classes do we need to construct the analysis classes? Figure 3.6 decomposes each of our analysis classes into groups of

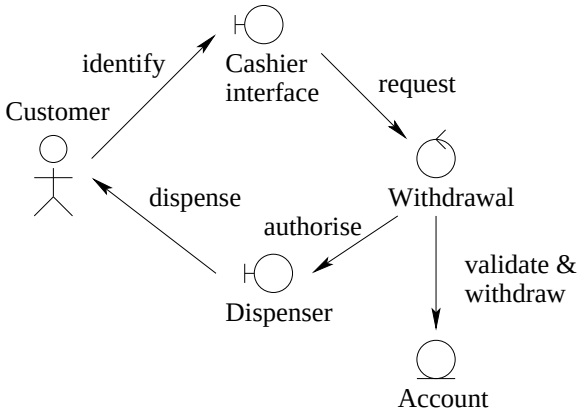

Figure 3.5: Collaboration model for withdrawing money

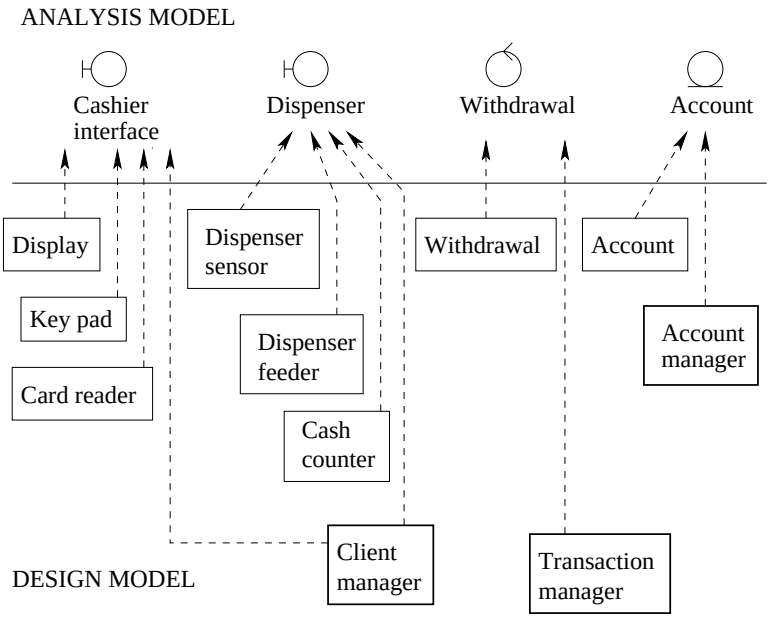

Figure 3.6: Design classes introduced for analysis classes

design classes. For instance, the cashier interface needs a display, keypad and card reader.

Our design classes will interact, as shown in the diagram of Figure 3.7 - for example the card reader obtains information from the customer and supplies information to the client manager. Notice how the interactions at this level are consistent with those anticipated in the collaboration model of Figure 3.5.

Finally (as far as this example is concerned) we consider the sequencing of events for interactions permitted in Figure 3.7. We depict this visually in the diagram of Figure 3.8. The temporal sequence of events is read from the top of the diagram and each directed link shows the interaction between design classes. Thus the sequence starts with the customer inserting a card in the card reader; then the card reader informs the client manager that the card has been inserted; then the client manager contacts the display to request a personal identification number; and so on.

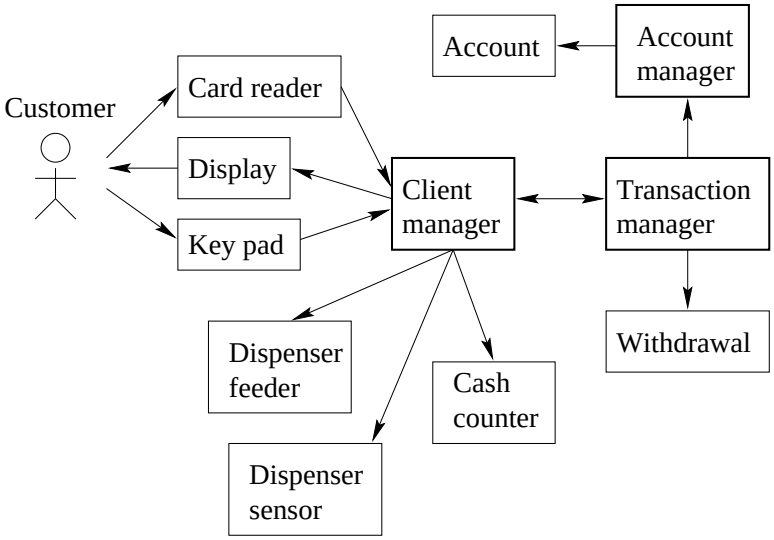

Figure 3.7: Interactions between design classes - part of design realisation

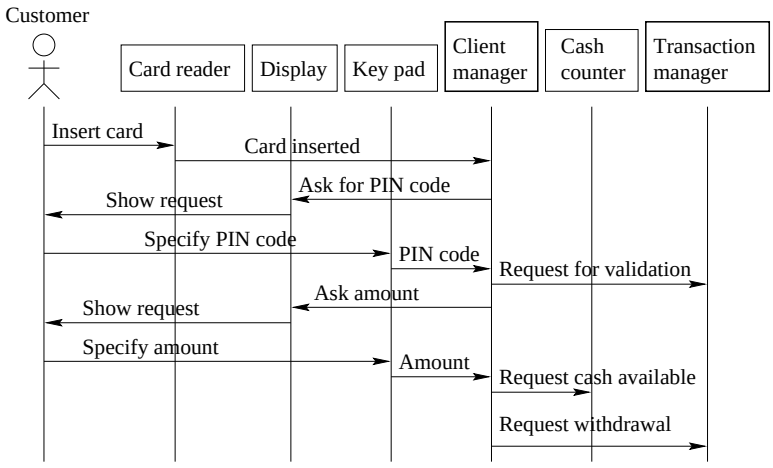

Figure 3.8: A sequence diagram for part of the realisation

Details in Jacobson, Booch & Rumbaugh *et.al.* 1998, *The Unified Software Development Process*.

3.2 Extreme Programming (XP)

Traditional "cathedral" methodologies (*e.g.* the Unified Process) concentrate on carefully controlled, up-front, documented thinking - we don't want to waste a lot of resources attempting to build a cathedral that was flawed even when its blueprints were written. The root cause of this concern is that when using these traditional methods the cost of unravelling bad decisions made in early design stages rises (perhaps exponentially) if they persist through later stages, so a significant mistake during requirements analysis will really hurt us if it persists to coding. Global control throughout is intended to minimise this risk.

But what if we could keep the cost of reaction to change in any aspect of the design nearly constant throughout the design lifecycle? That might free us from the restrictions (and risks) associated with traditional methods but would require some other form of discipline to keep the various design activities closely integrated. Extreme Programming is one way to do this.

Extreme Programming imposes discipline on design through twelve "practices" to which designers adhere (using whatever other compatible methods and tools they prefer). It is not strongly influenced by a particular design paradigm (like the Unified Process) but it does require a strongly held view of how to approach design. To give you a taste of this, here are a few of the twelve practices:

The Planning Process : The "customer" defines the business value of desired features. The programmers provide cost estimates for producing them in appropriate combinations. We are not allowed to speculate about producing a total system which costs less than the sum of its parts.

Small Releases : We put a simple system into production early, then re-release it as frequently as possible while adding significant business value on each release (*e.g.* Aim for monthly rather than annual release cycles). The aim is to get feedback as soon as possible.

Simple Design : Do the simplest thing that could possibly work. Don't design for tomorrow - you might not need it.

Testing : Focus on validation at all times and try to make validation as unbiased as you can, so write the test suites for your software before writing the software itself and make sure that your customers provide acceptance tests. All of this to take place within a rapid design cycle.

Refactoring : This method dives straight into coding, so re-design is vital. One heuristic for this is the "Three strikes and you refactor" principle -consider removing code duplication if:

- The 1st time you need the code you write it.
- The 2nd time, you reluctantly duplicate it.
- The 3rd time, you refactor and share the resulting code.

This needs a system of permissions for change between teams.

Pair Programming : All code is written by **a pair** of people at one machine.

- One partner is doing the coding.
- The other is considering strategy (Is the approach going to work? What other test cases might we need? Could we simplify the problem so we don't have to do this? *etc*).

This is unpalatable to some but has been claimed by practitioners to be vital to the method.

Collective Ownership :

Put a good configuration management tool in place. Then anyone is allowed to change anyone else's code modules, without permission, if he or she believes that this would improve the overall system.

Continuous Integration : Integration and testing happens no more than a day after code is written. This means that individual teams don't accumulate a library of possibly relevant but obscure code.

On-site customer : Someone who is knowledgeable about the business value of the system sits with the design team. This means there is always someone on hand to clarify the business purpose; help write realistic tests; and make small scale priority decision.

Coding Standard : Since XP requires collective ownership (anyone can adapt anyone else's code) the conventions for writing code must be uniform across the project. This requires a single coding standard to which everyone adheres.

See Kent Beck, 1999, *Extreme Programming Explained*.