1 Introduction

This program documentation is inspired by a very readable article written by Parnas and Clements: *A Rational Design Process: How and Why to Fake It* [1]

> *This paper brings a message with both bad news and good news. The bad news is that, in our opinion, we will never find the philosopher's stone. We will never find a process that allows us to design software in a perfectly rational way. The good news is that we can fake it. We can present our system to others as if we had been rational designers and it pays to pretend do so during development and maintenance.*

I will here record how I could have designed and written the Greed program as if I had known all the issues and all the answers beforehand. I will end with a short discussion of the benefits or otherwise of basing this program on the DCI paradigm.

Tom Love's program specification is quoted in section 2. In section 4, I consider the specification and decide on a plan of attack. The following sections will then describe the work as it could have progressed towards a working and readable program.

I did not know at the outset if this example will benefit from being written according to the DCI paradigm. I will end this report in SEC3 with a discussion of the DCI benefits or otherwise for this particular example.

2 Greed Specification

Ton Love sent the following specification to all workshop participants by fax:

> *Greed is a dice game played by two or more players. The object of the game is to tally points from the rolls of the die, and to be the first player to score 5000 points. There are five die in the game, which are rolled from a cup.*
>
> *To enter the game, a player must score at least 300 points on the first role of his turn, otherwise the player is considered "bust." If he goes "bust," he must wait until his turn to role again. If his first roll does produce 300 or more points, the player then has the option of stopping, thus keeping the initial score, or continuing. To continue, the player rolls only the die that have not yet scored in his turn. A player may continue rolling until all the dice have scored, or until he is "bust." With the exception of the entry roll, a "bust" is when an individual roll produces no points. The player may stop and keep his score after any roll, as long as he is not "bust."*
>
> *Each roll of the dice is tallied as follows:*
>
> > *Three of a kind score 100 x face value of one of the three die. If the three of a kind is 1s, then it is scored as 1000. 22234 = 200 points; 43414 = 400 + 100 = 500 points.*
> >
> > *Single 1s and 5s score 100 and 50 points, respectively.*
>
> *Examples (for first roll):*
> *44446 = 400 points, and the player would have the option to role the 4 & 6.*
>
> *11111 = 1000 + 100 + 100 = 1200 points, and the player has the option to roll all five die again.*
>
> *12315 = 100 + O + O + 100 + 50 points, and the player would have the option to role the 2 & 3.*

1.D. L. Parnas; P. C. Clements: *A Rational Design Process: How and Why to Fake It*IEEE Trans. on Software Engineering, SE-**12**. 2; February 1986

Finally, the winner is determined after a player collects a total score of 5000 or more, and all players have had an equal number of turns. If, for example, a player scores over 5000 points, he may still lose if a subsequent player ends up with a final score greater than his.

We see a simple game. Its only complexity is in its scoring rules. The main and probably only use case is to play the game in a number of rounds until finished.

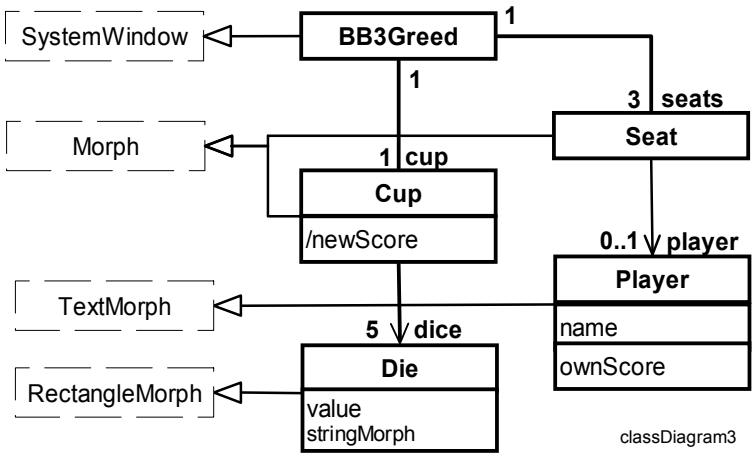
3 The User Interface

The user interface in the BabyGreed implementation is to be boringly simple. The interface is purely textual with no fancy graphics. Figure 1 shows a first version.

Figure 1: The user interface.

I make the game itself a SystemWindow, the other parts are TextMorphs and a RectandleMorph as shown in figure 2.

Figure 2: Making the Greed elements visible on the screen.



Next to bind it all together, furnishing the BB3Greed class with methods for configuring the interface and opening the window.

4 Project planning and system architecture

Three aspects of the problem need be considered. One is the user interface; it can be made very fancy with the dice and cup in 3-D and animated throws. Some submitters to the OOPSLA workshop had focused on this aspect and presented very impressive solutions. Another aspect is the player's strategy. At least one submitter implemented a very advanced game model.

A third aspect is the program architecture and implementation. This aspect is in the forefront of the current solution. The idea is to build a maximally clean and readable solution that can later be extended with fancy graphics and advanced mathematics.

I first have to choose my approach to the programming task. Test Driven Development (TDD) is one possibility. I reject it in its pure form because I believe that the data model is as important as the use cases so that data model and use cases need be considered separately. Also, I will not permit the code for the data model be contaminated with code for the use cases.

I will not provide an automatic test suite because I want to focus on making the code readable and will endeavor to make the system architecture obvious from the code.

The Agile Manifesto is all about how programmers work together. I am a one man team, so may be it doesn't apply here.

The static part of the specification looks simple and complete, so I decide to code the data model first and then add system behavior in a second step.

4. 1 The Data projection

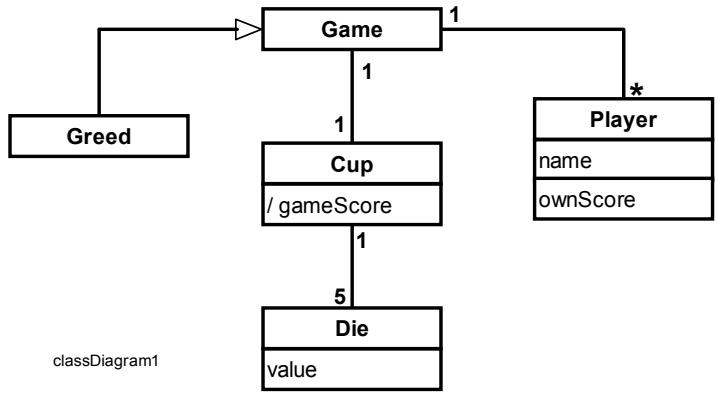
The first paragraph of the game specification tells us what the game *is*; the remainder tells us how it is *played*. The first specifies the static data model; the second specifies its dynamic behavior.

An old rule of thumb is to underline the nouns in the specification in order to identify candidates for data classes.

> *Greed is a dice game played by two or more players. The object of the game is to tally points from the rolls of the die, and to be the first player to score 5000 points. There are five die in the game, which are rolled from a cup.*

This first specification paragraph leads directly to the UML class diagram in figure 3

Figure 3: First UML class diagram.



Our task is to implement the Greed game, we are not asked to implement a general dice game.We start simple, and merge the Game and Greed classes.

An important point is that this class diagram does not show the dynamic structures that are needed to actually play the game. We will implement the dynamic structures in a Context class for each use case, leaving the data classes as simple as possible.

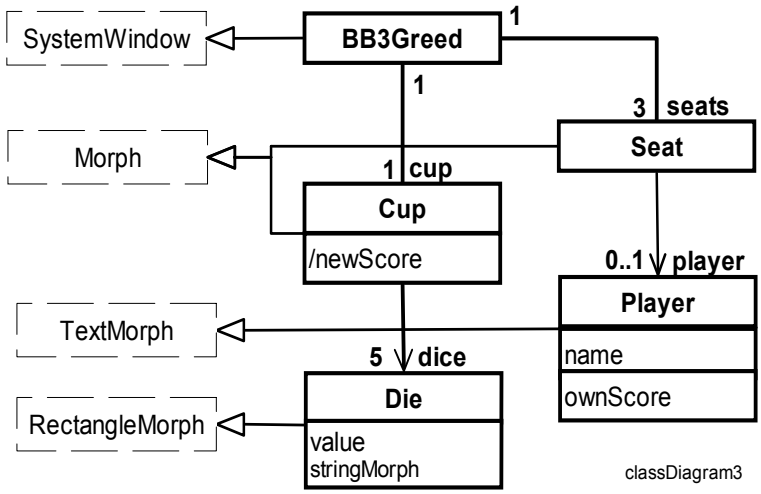
The next decision is about user interaction. The MVC paradigm has a clear separation between user interaction and domain data:

User	The most important participant is not represented in the acronym.
Model	The computer internal representation of the Greed game.
View	Makes domain data visible and tangible in a way that facilitates user interpretation of the model data.
Controller	Coordinates several views. (Note: The Controller is an input device in Smalltalk-80. There is no element coordinating several views).

MVC is useful when domain information need to be presented in different ways or in multiple copies on the screen. Here, there is no need for multiple views of the same information since we want the game of Greed be directly visible and tangible on the screen. We therefore decide to implement the game in Morphic with the Greed objects showing themselves on the screen.

I tentatively design a simple class structure for Greed and start by implementing the class diagram shown in figure 4. I decide to prefix all class names with *BB3* to distinguish them from all other class names.

Figure 4: First UML class diagram to be implemented.



I enter my Squeak image,

- open BabyIDE from the World menu *open...>>BabyIDE*.
  Answer *BB3Greed* when it asks for app name.
- The Greed class shall specify the handling of input to the game. The class shall therefore be defined in the *Env* perspective. Select the *Env* perspective.
- The usual text for a class definition is shown in the code pane.
  Fill in superclass *Morph* and class name *BB3Greed*. Accept.
- Select the *Data* perspective. Define the *Cup*, *Die*, and *Player* classes.

4. 2 The Use Cases

BB3AddPlayerCtxAdd player.

BB3PlayGameCtxPlay.