CSCI321 Project

D20 character generation
and maintenance

Preliminary Technical Manual

Autumn Session
May 2004

'Team d20'
Caleb Avery
Mark Boxall
Mark Hellmund
Mavis Shaw
Michael Tonini

Peter Castle

Table of contents

Introduction

Welcome to the *d20 character generation and maintenance program*. This powerful tool will help you to create character statistics for all d20 game sets, as well as allowing you to make running changes throughout the life of the character. With facilities that allow you to create a new d20 system character from scratch, edit an existing character or create a random character of your choice, the *d20 character generation and maintenance program* makes character creation and maintenance easy!

Role-playing games, also known as RPG's, are paper-based adventure games that take the players on journeys and campaigns filled with monsters, treasure and knowledge, and players must create their own characters to participate in these. Based on the d20 system, players go through a series of steps and rules to create all of the characteristics required to make a character with all of the necessary attributes needed. All of this information will be stored on an appropriate medium, such as a character sheet, for future reference. An example of a popular RPG is that of Dungeons and Dragons.

Throughout game play with the character, certain aspects of the characters change due to occurring factors such as battles and discoveries. Players must continually update their character sheets to keep up with these changes, which can prove to be difficult at times, especially if a character is 'levelling up'. Many attributes and statistics affect each other, so each change that is made needs to be checked to ensure it will not impact on another. Without the appropriate tools this process can be time-consuming, but unfortunately cannot be avoided and must be completed after each campaign the character participates in. Is there an easier way to do this?

Yes there is! The *d20 character generation and maintenance program* allows you to all of this quickly and easily, and requires no pen-and-paper work to do it. For those wanting to create a new character, the program conveniently splits character generation into 10 sections for easier working. The program ensures all entered values comply with the rule set of the chosen d20 game, informing you instantly if there is a problem. *D20 character generation and maintenance program* also allows you to edit saved characters for when changes to characters statistics need to me made. Also with the built-in functionality of random character generation for a quick alternative to character creation, all of your character needs are catered for in this program.

The *d20 character generation and maintenance program* can be used for all of the popular d20 system role-playing games, including Dungeons and Dragons and d20 modern.

Project description

Title:
D20 character generation and maintenance

Description:
Role-playing games such as Dungeons and Dragons have a very complex rule set for character generation, progression and storage. The aim of this project is to create an application that embodies the current rules, plus the facility to adapt to future rule changes, that will allow the easy creation and tracking of D&D character statistics.

Extensibility to other d20-based games would be an advantage, as would other Game Administrator tools.

Tools and resources:
Operating system: any
Language: C++, no special hardware.

CSCI321 is a third year computer science subject at the University of Wollongong where a group of students work together to design and develop a computer program from initial conception to final implementation and presentation. It is an annual subject, running over two full sessions, and is normally undertaken during 3rd year studies for computer science and information technology students.

Normally, academic staff at the university that may require a system to be developed create problem descriptions for students to work with, and supervise their progress throughout the project. However, the members of *'team d20'* created and drafted a project description proposal in relation to character creation within the d20 role-playing game system. Once a suitable supervisor agreed to work with the group, the proposal was accepted as an official project.

The team is comprised of five members and one supervisor. Caleb Avery, Mark Boxall and Mark Hellmund are current third year computer science students, while Mavis Shaw and Michael Tonini are third year information and communications technology students. Peter Castle is a lecturer within the school of information and communications technology at UoW, and has offered to supervise the project. With his extensite knowledge of many RPG games his input has been invaluable.

With everyone within the group having prior knowledge of each member, the group dynamics have been strong from the start, enabling the project group to move quickly to first learn from scratch the rules of the d20 game Dungeons and Dragons, and then design and develop an effective and efficient software system that will aid players in the creation and maintenance of new and existing game characters.

Traditional character generation

Dungeons and Dragons is a d20 (20-sided die) based roll playing game (RPG). The following is a step-by-step guide to producing a 1st level character manually for playing D&D.

Prerequisites:

- Players Handbook v.3.5 (PHB),
- A photocopy of the character sheet (Found in the back of you PHB),
- A pencil
- A scrap of paper
- Four 6-sided dice

Step 1: Check with your Dungeon Master

Check with your Dungeon Master (DM) about any house rules, and how to apply them to the standard rule set. Also find out what other members of your party have created, so that your character fits the dynamic of the party.

Step 2: Roll Ability Scores

Roll you characters six ability scores by rolling four six-sided dice. Ignore the lowest scoring die and total the three highest scoring dice. Record the six totals on your scrap of paper.

Step 3: Choose Class and Race

Choose both your character's class and race at the same time, as some classes better suit some races. (Ref. PHB 7-60.)

Step 4: Assign and adjust Ability scores

Once you have chosen your character's class and race, use the ability scores you rolled in step one and assign each to one of the six abilities: Strength, Dexterity, Constitution, Intelligence, Wisdom, and Charisma. Adjust your characters scores up or down, according to your race. (Ref. PHB 12.)

Step 5: Record Racial and Class Features

Your character's race and class provide certain features, such as feats and skill bonuses. Most of these are automatic, but some involve making choices and thinking ahead about upcoming character creation steps.

Step 6: Select Skills

Your character's Intelligence modifier determines how many skill points you have to 'buy' skills with. (Ref. PHB your Class). Your can buy your character's level + 3 skill ranks. So at 1st level you can buy 4 skill points in a non-cross-class skill, and 2nd level you can only have 5 ranks total in that same skill.

Step 7: Select Feat

Each 1st-level character starts with one feat. Page 90 of the PHB lists all feats, their prerequisites, and a brief description.

Step 8: Select Equipment

To randomly determine your starting gold, see page 11 of PHB. Once you have calculated this, you can then buy your own gear piece by piece, using the information from Equipment chapter of the PHB.

Step 9: Record Combat Numbers

Determine these statistics and record them on your character sheet.

Hit Points: Your hit points (HP) determine how hard your character is to kill. At 1st level, wizards and sorcerers get 4 HP; rouges and bards get 6 HP; clerics, druids, monks, and rangers get 8 HP; fighters and paladins get 10 HP; and barbarians get 12 HP. To this number, add your character's Constitution modifier.

Armor class: Your Armor Class (AC) determines how hard your character is to hit. Add the following numbers together to get your AC: 10 + your armor bonus + your shield bonus + your size modifier + your Dexterity modifier.

Initiative: Your character's initiative modifier equals your Dexterity modifier.

Attack Bonuses: Your class determines your base attack bonus. To determine your melee attack bonus for when you get into close combat fights, add your Strength modifier to your base attack bonus. To determine your ranged attack bonus for when you attack from a distance, add your Dexterity modifier to your base attack bonus.

Saving Throws: Your class determines your base saving throw bonuses. To these numbers add your Constitution modifier to get your Fortitude save, your Dexterity modifier to get your Reflex save, and your Wisdom modifier to get your Will save.

Step 10: Details

Now choose a name for your character, determine the character's gender, choose an alignment, decide the character's age and appearance and so on. (Ref Chapter 6: Description of PHB).

Elements of D&D

Abilities

There are six main abilities - Strength, Dexterity, Constitution, Intelligence, Wisdom and Charisma. Each of the abilities is a numerical representation for how strong you are, how smart you are etc. Depending on how high or low the values are for each, you will recieve ability modifiers. These give you bonuses when performing certain actions. For example a stronger fighter will do more damage, as there is more power in each hit as compared to a weak individual, who would do less damage than the average person. The average for every attribute is 10. For every 2 ability points you are above or below, you take a bonus or penalty accordingly. Abilities are determined when your character is first created, then you select your class and race, then assign the highest to the appropriate ability for that class/race.

Races

D&D is a game of the imagination, as such it has multiple races in its' worlds. There are 7 races; a character can be human, dwarf, elf, gnome, half-elf, half-orc and halfling. These all affect the ability scores, for example a Dwarf has increased constitution and decreased charisma. This has an effect on what class each race is suited for, for example, a dwarf is not very charismatic, which means he is unable to negotiate as well, making people dislike him more than other races. However, he can take more hits due to his stronger constitution. This is why a dwarf is suited to be a fighter.

Classes

A character class is a chosen job/profession; with eleven choices available covering many character traits and characteristics. You can be a barbarian, bard, cleric, druid, fighter, monk, paladin, ranger, rogue, sorcerer or wizard. Different classes allow players to gain more ranks in class skills (a rogue, a born thief, can have more skill points into sleight of hand at each level than a ranger), and also what feats and skills they gain as they progress through levels. It also determines their base attack bonus, base save bonus and the maximum number of additional hit point they can gain per level.

Skills

Skills represent actions, training and disciplines in which the character is proficient. His ability to perform these at varying complexities is effected by how many skill points have been allocated to each skill – these are assigned when the character is created and are governed by their class and intelligence. Characters of certain classes and races will be more capable at certain skills - in fact some skills are class skills that give them bonuses to progress faster in them taking less skill points. For example, a wise character is better at healing, as they are more knowledgeable. A cleric can reach a higher level of healing at a lower level, as it's a class skill. Every skill has an ability where a modifier will increase or decrease its amount when a skill check is made.

Feats

Feats are very much like skills - they give a character a new ability or merely improve a skill/modifier. Unlike a skill however you do not gain levels in feats – you either have it or you don't. Feats can also have pre-requisites. Characters get one choice of a feat when they are created, then one every 3rd level thereafter.

Character description

Every character has a different physical appearance, different age, pray to a different god and have different reasons for becoming the character he or she is. Characters also have an alignment, which will determine how they act. The alignments are Lawful Good, Lawful Neutral, Lawful Evil, Neutral Good, True Neutral, Neutral Evil, Chaotic Good, Chaotic Neutral and Chaotic Evil. For example, a Chaotic Evil character will do anything that benefits them regardless of the after-effects. Whereas a Lawful Good person will uphold the law and help people at their own expense. Depending on their alignment, characters worship a god that follows that alignment. When a character is created all of the characters description details are selected, however only alignment has a direct impact on the game. A background story and looks are only for the players benefit for role-playing.

Equipment

As a character progresses they make money in the form of coins. These are in the form of copper, silver, gold or platinum pieces, which they can barter for items. Characters have armor, weapons, magical items and many miscellaneous items. Each character starts off with a different set of items depending on their class. They will also receive different amounts of starting gold, depending on their chosen class. Any modifiers will only apply to items that the character currently has equipped. Smaller creatures can only use smaller items. Characters also have weight limits depending on their strength and size – if a characters carries too much they become encumbered and move slower.

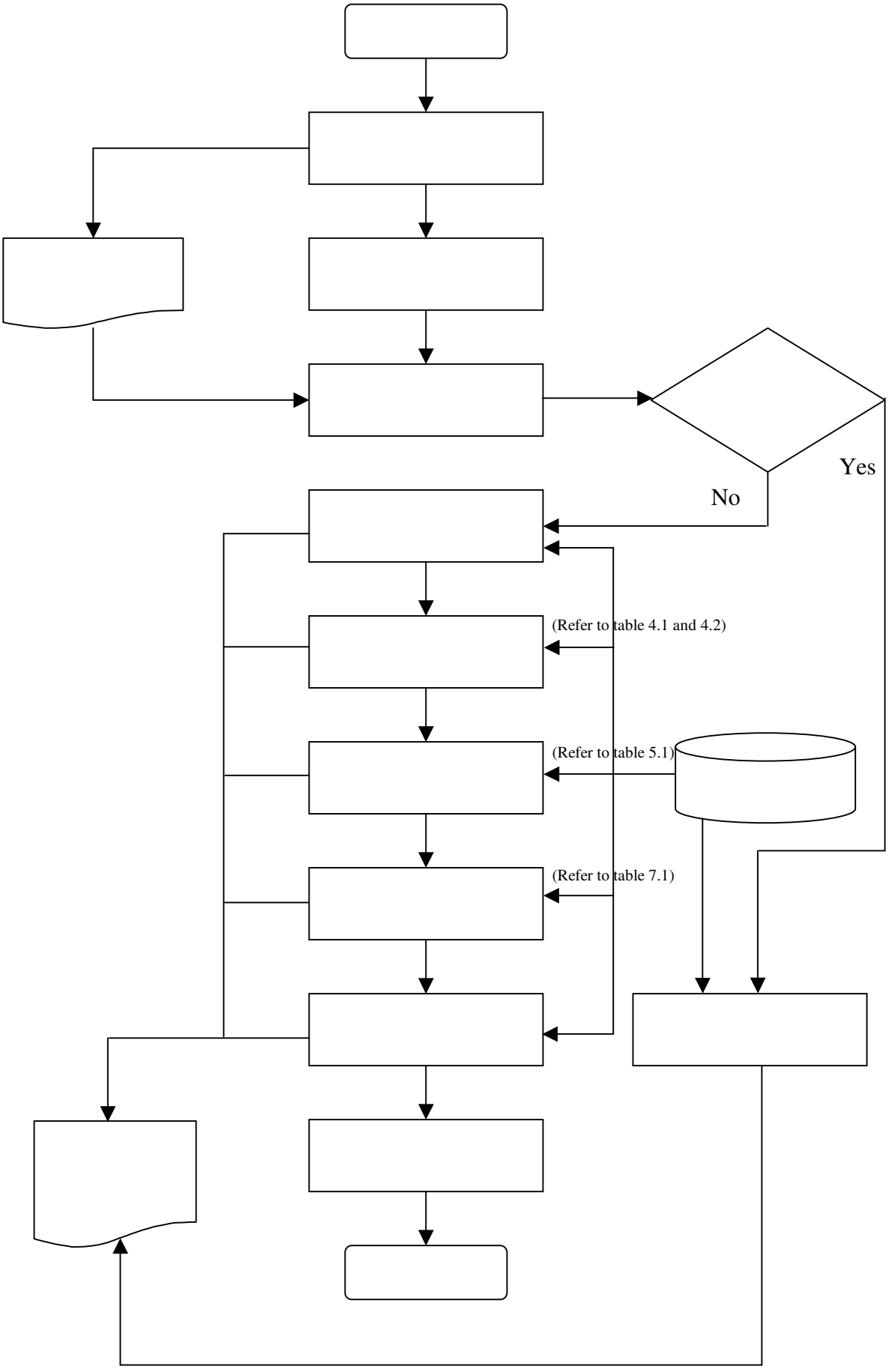
Combat

Combat in D&D is turn-based – everybody has a chance to choose their action. The order of play is decided by who has the highest initiative, this is determined by who rolls the highest number in addition to their dexterity bonus. Characters then go in the order of who attained the highest number. People then can roll for all of these actions. If their Attack Roll is higher than the monsters Armor class roll they are attacking, they score a hit and roll again for damage done. A person can use any skills they have in combat as well as perform attacks with their equipped weapon, as long as the action only uses one round. One round is determined as 6 seconds in the game world. Spells can also be cast within a round.

Adventuring

Every adventure has to start somewhere, which is usually not where it began. Journeying and exploring is a large part of D&D playing. This involves either walking, hiring a transport, riding or any of the other possibilities the player or Dungeon Master can think of. Players can go anywhere – dungeons, towns, mountains or caves – the possibilities are only limited by the Dungeon Masters imagination. Upon completing missions, treasure or payment is usually found which is divided up evenly between party members. Characters can also gain things like reputations, followers, land and titles or honours.

Character creation process



This flowchart provides a visual representation of the character creation process. There are two main sources of input/output (D&D Rulebook and Character sheet). There is also a temporary input/output to record ability rolls until they are assigned.

Processes:

- *Roll Ability Scores* – Ability scores are rolled and temporarily recorded.
- *Choose Class and Race* – Choose class and race and record them to Character sheet.
- *Assign and Adjust Ability Scores* – Based on class an race assign the ability scores to the most appropriate ability.
- *Record Racial and Class Information* – Make choices on racial and class information and record it.
- *Select Skills* – Add ranks to skills that are chosen.
- *Select Feats* – Select a feat.
- *Calculate Wealth and Select Equipment* – Roll according to class your gold amount, then select starting items.
- *Calculate Combat Numbers* -
- *Record Details*

For an in-depth description of each of the processes refer to the traditional character generation section and review the process with the corresponding title.

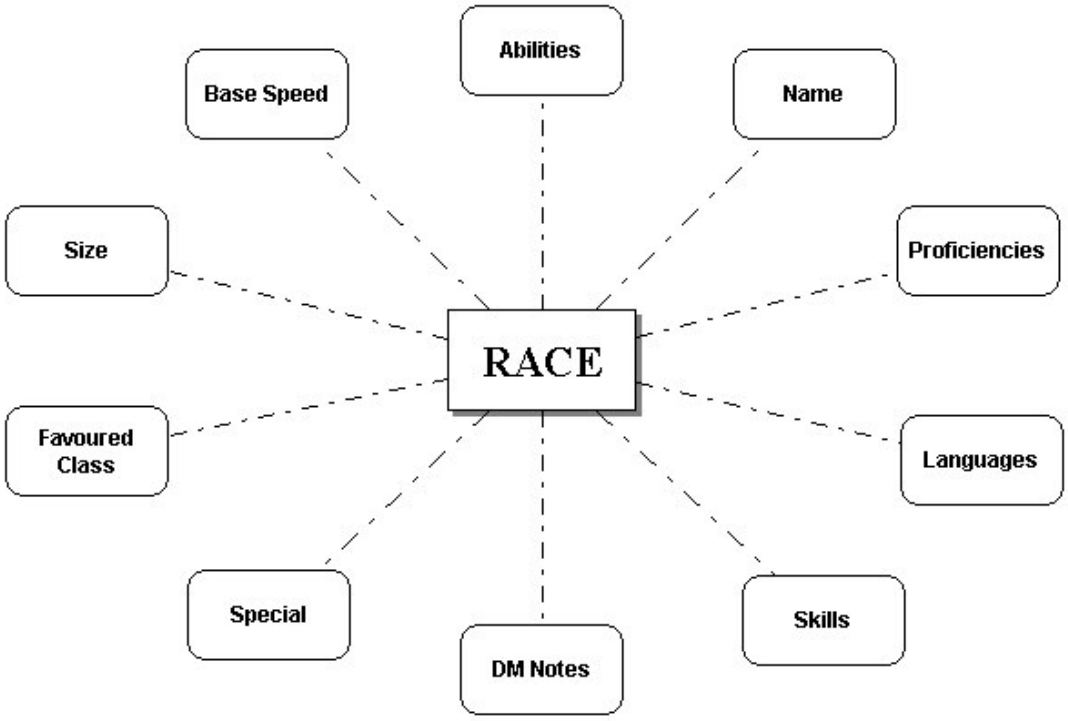
The starting package decision allows many details to be used from a predefined set instead of having to move through several of the later processes.

The processes modelled here are broken down in later sections.

Visual breakdown with mind maps

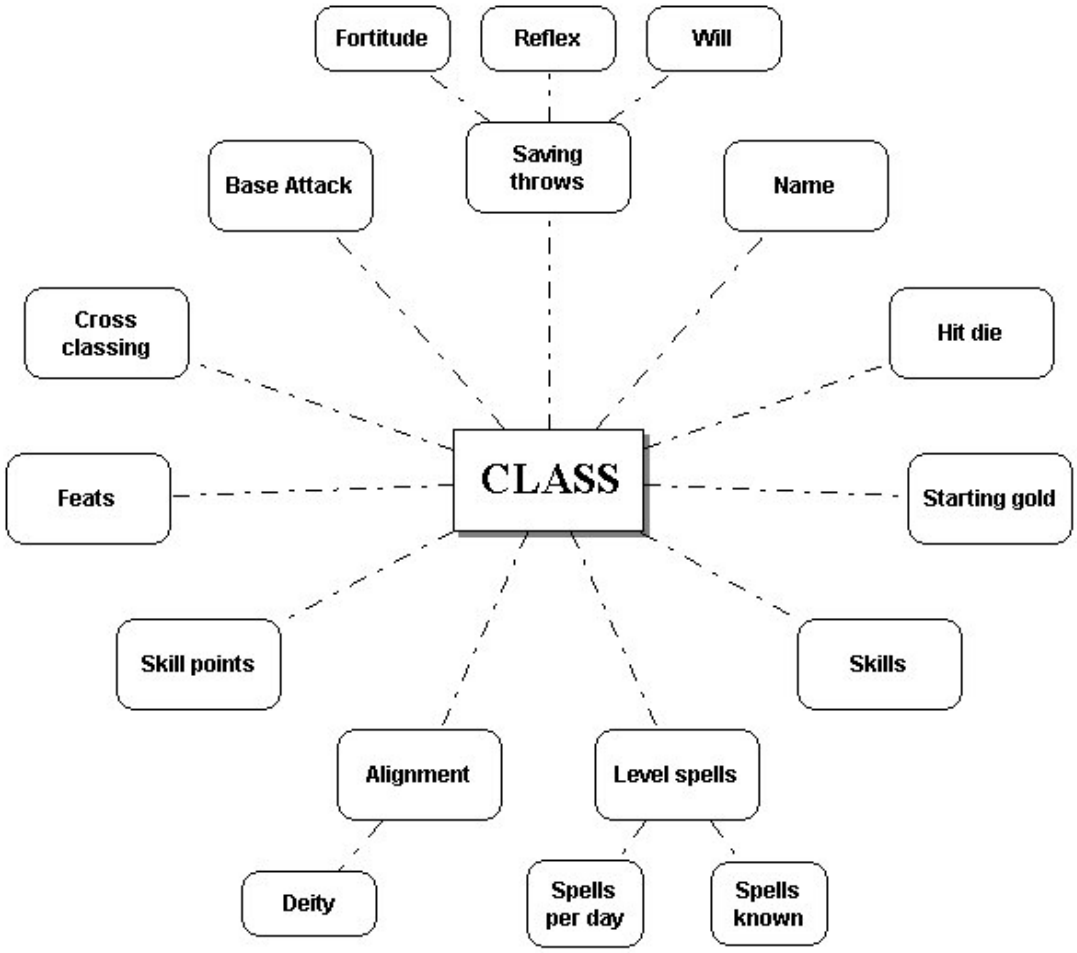
To gain a greater understanding of the elements within the Dungeons and Dragons game, mind maps were created for all of the important classes to assist in designing class structures. Through deciphering the rules the following mind maps were created, forming the basis for the attributes that would be stored within each program class. A brief description of each displayed attributes is also included.

Race



Name:	Name of the race;
Abilities:	The 6 core game elements – Strength, Dexterity, Constitution, Intelligence, Wisdom and Charisma;
Base Speed:	Moving speed of race at normal weight;
Size:	Size of the creature – tiny, small, medium, large, giant, etc;
Favoured Class:	Character class that will most suit the chosen race;
Special:	Anything special about the race that could not be categorized;
DM Notes:	Any racial information the DM will need to know:
Skills:	Skills that the race automatically gives the player;
Languages:	Languages the race knows, as well as languages they can learn;
Proficiencies:	Special circumstances where a race will get proficiencies in particular items, i.e. weapon proficiency levels may be modified for certain items.

Class



Name: Name of class,
Hit Die: Die used to determine attack damage,
Starting Gold: Amount of gold allocated to character at creation,
Skills: Skills that the class automatically gives the character,
Base Attack: Minimum amount of damage caused in an attack,
Cross-classing: Whether the class is affiliated with another class,
Feats: Feats that the class automatically gives the character,
Skill Points: Number of points class can allocate to their skills,
Saving throws: Classes ability to withstand certain attacks, either good or bad,

- Fortitude: physical stamina,
- Reflex: agility or quick responses,
- Will: mental thoughness,

Alignment: Combination of attributes that determine behaviour – good, evil, lawful, chaotic, neutral,

- Deity: god of worship associated with alignment,

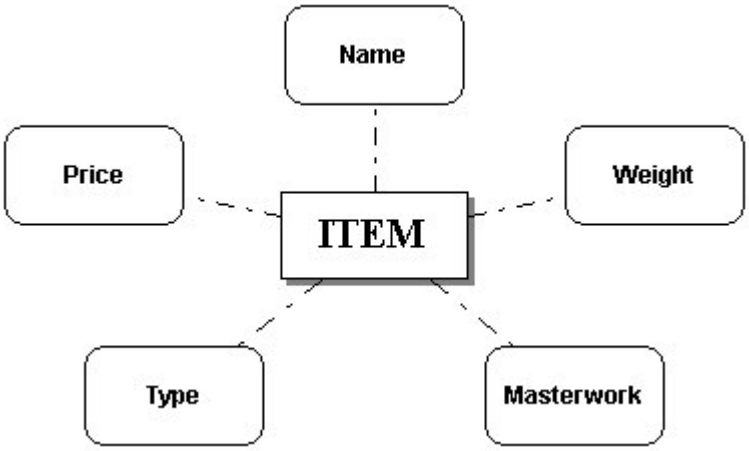
Level spells:

- Spells per day
- Spells known

Item

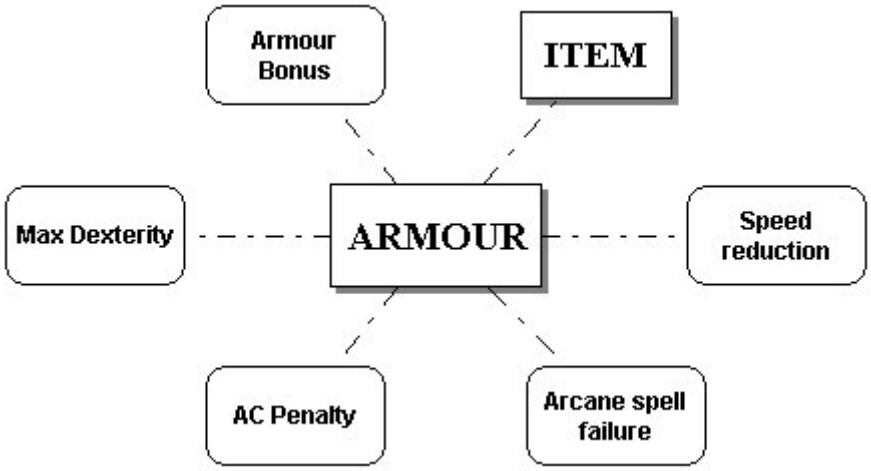
The item section proved to be an interesting one, due to the many different types of items, and how differently some of them interact with other items or characters. If all of the attributes required to cover all relevant item information were to be stored within one structure, it would be extremely inefficient due to the variation in different items.

To overcome this, a heirachy system was created where items requiring specific information to be stored about them will have a sub-class. Branching from the main items mindmap are the categories of armor, weapons and magic items. Within the items class, a flag will be set that will allocate a 0, 1, 2 or 3 depending on the type of item.



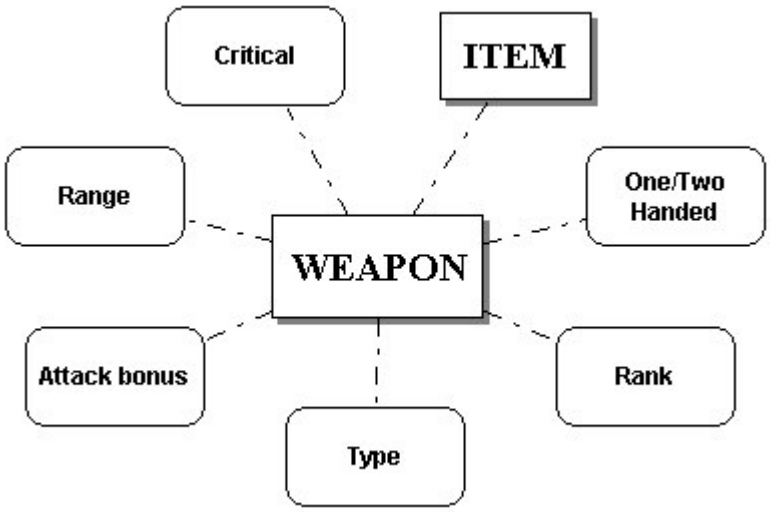
Name:	Name of item,
Weight:	Total weight of item in pounds (lb.),
Price:	Cost of item to purchase
Type:	Type of item, 0 for normal, 1 for armor, 2 for weapon, 3 for magical item
Masterwork:	Flag indicating whether item is aditionally masterworked.

Armor (inherits from Item)



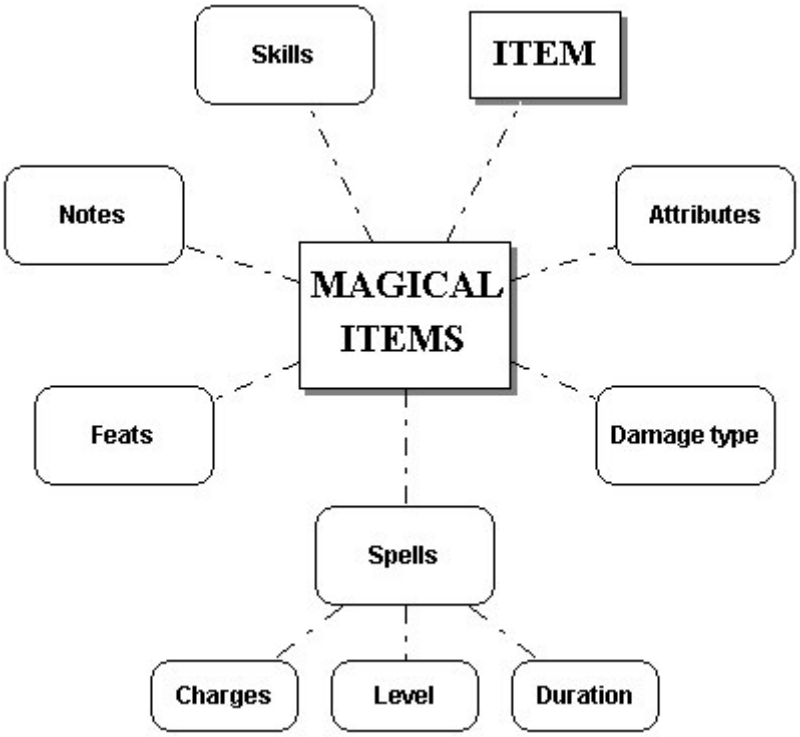
Armor bonus:	Any additional bonuses gained through armor,
Max Dexterity:	Maximum amount of Dex allowed by wearer,
AC Penalty:	AC reduction to characters attack due to weight,
Arcane spell failure:	Percentage chance of spell failing due to armor,
Speed reduction:	Maximum speed character can move wearing armor.

Weapon (inherits from Item)



Critical:	Critical roll modifier value,
Range:	Distance weapon is functional to,
Attack bonus:	
Type:	Main weapon category i.e. piercing,
One/two handed:	Whether one or two hands are required to use the weapon,
Rank:	Indicates skill level required to use – 0 represents simple, 1 represents martial and 2 represents exotic.

Magical items (inherits from Item)



Skills: Skills affiliated with the magical item,
Attributes: Attributes affiliated with the magical item,
Damage type:
Feats: Feats affiliated with the magical item,
Notes: Any special notes about the items functionality,
Spells: Spells affiliated with the magic item,

- *Charges:* Number of spell instances remaining,
- *Level:* Caster level required to use magical item spell,
- *Duration:* Length of time spell lasts.

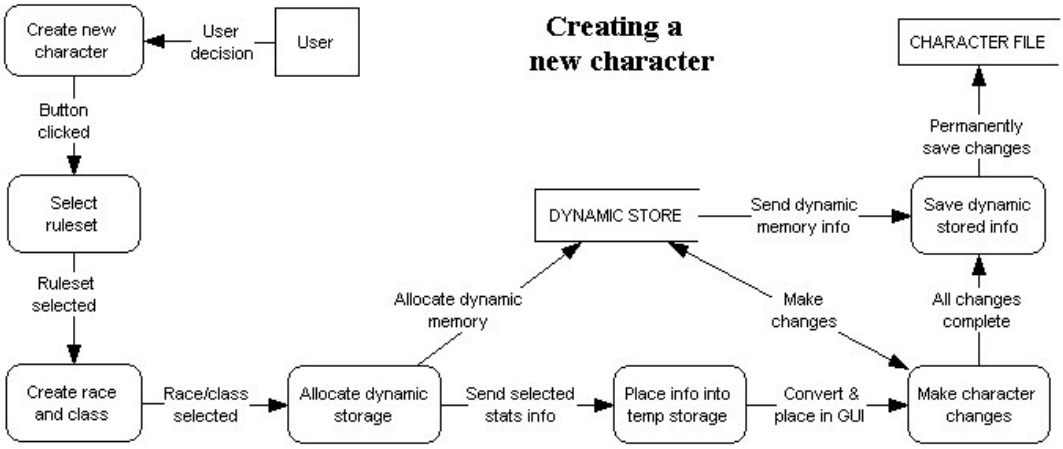
Program data process

Understanding the processes that will be undertaken in the final system will aid in the overall design process. After deciding what functions the program would perform, processes for each section were defined and converted into data flow diagrams for easier viewing and layout.

There will be three main processes within the program, which include:

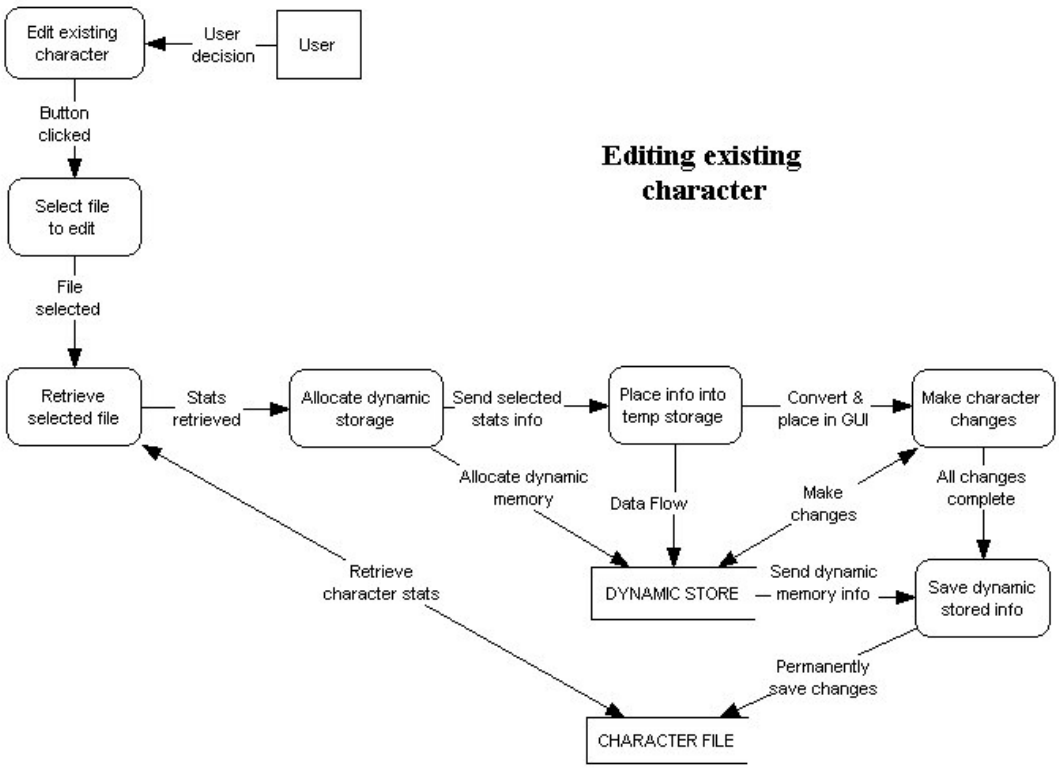
- Creating a new character
- Editing an existing character
- Random character generation

Creating a new character



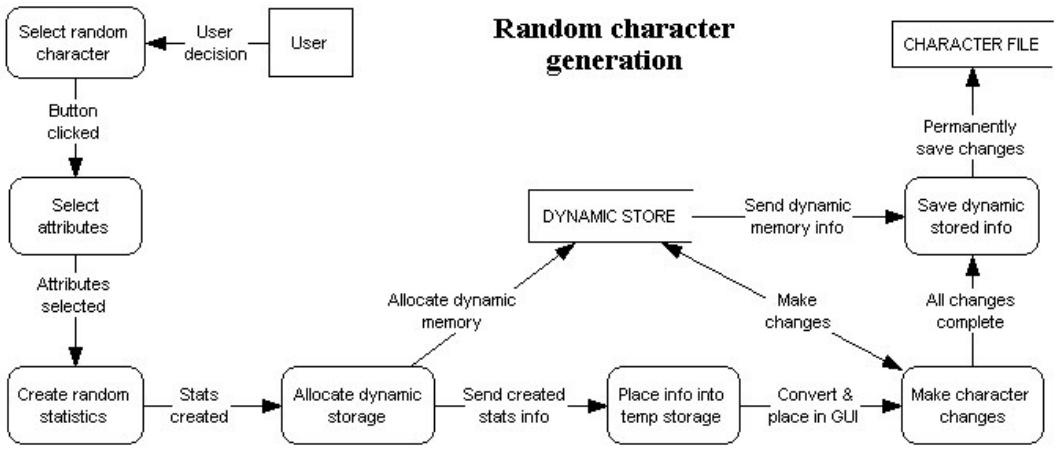
- *Create new character* – User selects the program to create a new character,
- *Select ruleset* – Once selected, user chooses the ruleset they wish to use,
- *Create race and class* – Once selected, user chooses their characters race and class,
- *Allocate dynamic storage* – Once base character information is selected, new dynamic storage is allocated to store the information,
- *Place info into temp storage* – Once created, the base character information sent from earlier operations is stored within the dynamic storage, and then placed within the GUI for user manipulation,
- *Make character changes* – User makes appropriate character statistic modifications to information displayed within the GUI, which is continually updated within the dynamic store,
- *Save dynamic stored info* – Once all changes are complete, the information stored in the dynamic storage is converted into a hard copy of the character file.

Editing an existing character



- *Edit existing character* – User selects the program to edit an existing character,
- *Select file to edit* – Once selected, user finds and selects file to open,
- *Retrieve selected file* – Once selected, the character file is located and opened, ready to read in the characters information,
- *Allocate dynamic storage* – Once character information is retrieved from the file, new dynamic storage is allocated to store the information,
- *Place info into temp storage* – Once created, the base character information sent from earlier operations is stored within the dynamic storage, and then placed within the GUI for user manipulation,
- *Make character changes* – User makes appropriate character statistic modifications to information displayed within the GUI, which is continually updated within the dynamic store,
- *Save dynamic stored info* – Once all changes are complete, the information stored in the dynamic storage is converted into a hard copy of the character file.

Random character generation



- *Select random character* – User selects the program to generate a random character,
- *Select attributes* – Once selected, user chooses the required character attributes to create the new random generated character,
- *Create random statistics* – Once the required attributes are selected, the program creates a set of random character statistics that match the chosen attributes,
- *Allocate dynamic storage* – Once the random generated character information is created, new dynamic storage is allocated to store the information,
- *Place info into temp storage* – Once created, the base character information sent from earlier operations is stored within the dynamic storage, and then placed within the GUI for user manipulation,
- *Make character changes* – User makes appropriate character statistic modifications to information displayed within the GUI, which is continually updated within the dynamic store,
- *Save dynamic stored info* – Once all changes are complete, the information stored in the dynamic storage is converted into a hard copy of the character file.

Classes and implementation breakdown

In order to store all of the data relevant to a D&D character in our program, a number of classes have been created to assist in the storage and manipulation of the data.

To limit the amount of hard-coded variable names and values, a mostly generic structure was created to allow for the reading of variable names and values from files. This has the added advantage of being able to use the potential variable name as an array key. Using class functions to reference values by an input string has greatly simplified the functions and allowed for a more generic approach.

Value modifier list

Another prominent idea central to our approach has been the ability to keep track of changes to values, so as to give the player a greater sense of understanding as to where all of the bonuses they have received have come from. In order to do this, a simple linked list will be used to add a "modifier" rather than adjust the value directly. The class takes care of adding up the modifiers and returning the total value, as well being able to return a description string that lists each modifier and its value.

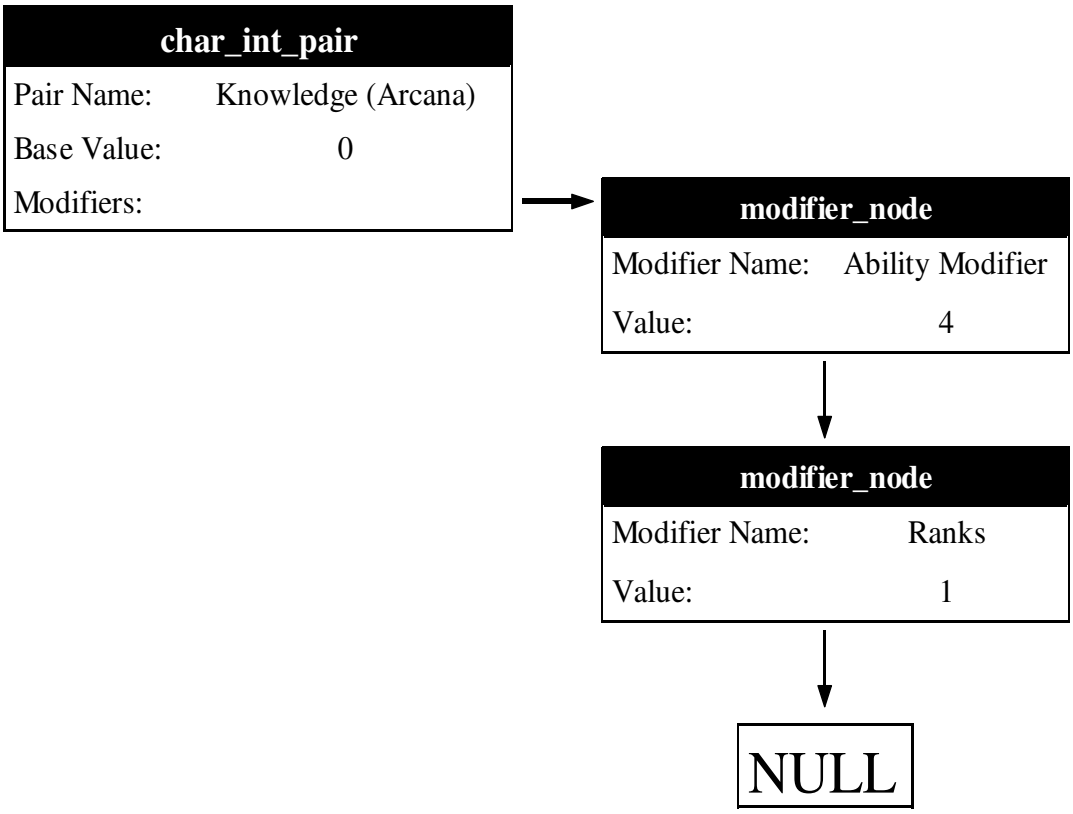


Diagram demonstrating the use of a linked list to store modifiers

A key structure in the list is the *modifier_node,* which both stores the modifier data and links to the next such node. The struct contains a pointer to dynamic memory and it is up to the class using the struct to control the allocating and de-allocating of that memory.

```
struct modifier_node
{
   char* modifier_name;
   int value;
   node* next;
};
```

The *modifier_node* contains a character string *modifier*_name used both as a key name in identifying the modifier and as a descriptive text to use when displaying the list of modifiers.

Also included in the node is the *value*, which may be either positive or negative depending on whether the modifier is a bonus or a penalty.

The last item in the node, *next*, is a pointer used to indicate the next node in the linked list of modifiers. If *next* is a NULL pointer, then the end of the list has been reached.

The *char_int_pair* class provides the overlying structure to the linked list, and is in charge of adding, removing, and changing modifiers in the list. As there is a lot of dynamic memory, this particular class is quite vulnerable to memory leaks if the proper checks are not made.

```
class char_int_pair
{
   private:
      char* pair_name;
      node* modifier;

   public:
      char_int_pair();
      char_int_pair(char*);
      ~char_int_pair();

      void set_name(char*);

      int get_total();
      char* get_explanation();

      void add_modifier(char*, int);
      void remove_modifier(char*);
      void reset_modifiers();
      int get_modifier(char*);
      void set_modifier(char*, int);

      int operator[] (char*);
};
```

The *pair_name* is a character string which also is used as its identifier.

The pointer *modifier* is the head of the linked list, and as such also defaults to a NULL value when first declared, and whenever the modifiers are reset.

There are two constructors in the class definition, a default constructor and an initialisation constructor. The default constructor will set both pointers to NULL, where as the initialisation constructor will set the *pair_name* to the given char* while again setting the *modifier* to NULL. The *set_name* function allows for changing the *pair_name* value after initialisation.

As there is a large amount of dynamic memory, a destructor was added to remove it all. This involves stepping through the list and removing any dynamic memory from there as well as removing the memory for the *pair_name*.

The *get_total* function steps through the modifier list and returns the cumulative total of all the entries in the list. The *get_explanation* function, again steps through the modifier list, but instead keeps a record of the individual modifiers by way of a character string. Displaying this string would list all of the modifier – value pairs.

The *add_modifier*, *remove_modifier* and *reset_modifiers* functions are all rather straight forward, adding the given modifier – value pairs, removing a modifier with a particular name, and removing all of the modifiers from the list. The function *set_modifier* allows the changing of a modifiers value, without having to remove and add again.

The *get_modifier* and *operator[]* functions are basically the same, return the value for a given modifier, but the ability to use the square brackets may make it easier to understand.

Experience class

The experience class is a very simple class which was only intended so as to control any changes to XP from within its own class. This is mainly preventative of changes made from other functions accidentally.

```
class xp
{
   private:
      long xp;

   public:
      xp();
      xp(long);

      void adjust_xp(long);
      long get_xp();
};
```

The experience points (XP) are stored as a long integer because it is understood that the actual XP value can rise to quite a large number.

The default constructor will simply initialise xp to a zero value, while the initialisation constructor will obviously initialise the value to the long integer given as a parameter.

The *adjust_xp* function (as expected) adjusts the value of xp by the given amount. This includes adding to or subtracting from depending on whether the amount is positive or negative. The *get_xp* function simply returns the value of xp.

Abilities, skills and combat classes

Due to the similarity of each the Ability, Skill and Combat classes, it is easier if a single class can be used for all of them. These classes are all simply a list of name-value pairs, and all of them would benefit from the addition of the modifier linked-list. The only difference is the names of the name-value pairs and the modifiers that are applied by them, and they can all be read from a file.

```
class char_int_group
{
   private:
      char* group_name;
      char_int_pair* dynamic_pairs;

   public:
      char_int_group();
      char_int_group(char*);
      ~char_int_group();
      void read_file(char*);

      int operator[] (char*);
      int operator[] (int);
      char_int_pair* operator[] (char*);
      char_int_pair* operator[] (int);

      void add_modifier(char*, int);
      void apply_modifiers();
};
```

The *group_name* like the other classes is used as an identifier by other classes, and is a dynamically stored character array. The name-value pairs are dynamically stored in the *dynamic_pairs* pointer.

There are two constructors, the default constructor to nullify everything, and the initialisation constructor which takes the name of the definition file as a parameter. The function *read_file* does the same thing as the initialisation constructor, but is used in conjunction with the default constructor if the definition file is not known at the time of declaration. The destructor is present to clean up all of the dynamic memory, because there are a lot of dynamic structures in this class.

The overloaded operators are used to return either the char_int_pair or the grand total of the associated modifier list. Both character strings and numbers are accepted as parameters to allow reference specifically by name or simply by number if stepping through the list.

The *add_modifier* function takes into consideration which pair to add a modifier to as specified in the char* parameter, removes the pair reference from the char* and sends the remainder to the pair's own *add_modifier* function.

The *apply_modifiers* function reads from a file the modifiers it should add to other groups and pairs, and instigates the process of applying them. By applying all the modifiers at once (checking requirements along the way) it minimises the possibility of adding the same modifier more than once.

Money

The addition and subtraction of money may be trivial like the previous experience class, but money can take many forms of currency and those currencies may change as may the difference in value between them. For this purpose a class was devised to deal with different forms of currency.

```
struct unit_currency
{
   char* name;
   int value;
   long amount;
};
```

The *unit_currency* structure is used in conjunction with the *money* class to allow varied currencies and exchange rates between the currencies. In original Dungeons and Dragons Rules v3.5 there are four forms of currency, platinum, gold, silver and copper. Each form of currency would have it's own *unit_currency* struct instance.

The *name* would be used to describe the currency, e.g. "gold", "gold pieces" or "gp". The *value* indicates the number of these units is necessary to make up a unit of the next most valued form of currency. The *amount* is the number of these units currently in the possession of the players character.

```
class money
{
   private:
      unit_currency* money;

   public:
      money();
      money(char*);
      ~money();
      void read_file(char*);

      void adjust_money(char*, long);
      long operator[] (char*);
      long operator[] (int);
      char* operator[] (int);
};
```

The amount of money is stored dynamically in a *unit_currency* pointer called *money*.

The default constructor sets *money* to be NULL, while the initialisation constructor takes a char* parameter which is the name of the file describing each of these currency types. The function *read_file* is there to add the same ability to an instance that uses the default constructor. The destructor is present to clean up any dynamic memory.

The function *adjust_money* takes a char* for the name of the unit currency and a value with which to adjust it by. The overloaded operators return the *amount* associated with a currency unit referenced by either name, or number. An operator also exists to return the *name* of a currency unit by number.

Physical description

In order to store the basic physical data about the character a simple class was developed to store all the information in some dynamically allocated character strings.

```
class physical
{
   private:
      char* char_name;
      char* player_name;
      char* gender;
      char* height;
      char* weight;
      char* hair;
      char* eyes;
      char* skin;
      char* age;
      char* god;
      char* alignment;

   public:
      physical();
      ~physical();

      void set_char_name(char*);
      char* get_char_name();
      .
      .
      .
      void set_alignment(char*);
      char* get_alignment();
};
```

All of the variables of this class do not need any explanation. The variable names are descriptive enough about the values they contain.

A constructor and destructor are present as always to nullify pointers on initialisation and to clean up the dynamic memory afterwards. In addition to the constructor and destructor, a pair of functions exist for each char* variable in the class. These are to give a value to the individual variable, and to return a value for the variable.

Items class

Items play a strong part in any game of Dungeons and Dragons. There are four main types of items; general items (clothing, books, etc), armor (chain mail), weapons (sword, cross-bow), and magical items (wand, ring or protection). All these different types of item have different properties and hence require special variables (a sword will behave differently to a torch).

```
class item
{
   private:
      char* name;
      int weight;
      bool masterwork;
      int type;

      int price;
      char* currency;

   public:
      item();
      ~item();

      void set_name(char*);
      char* get_name();
      .
      .
      .
      void set_currency(char*);
      char* get_currency();
};
```

The normal *item* class contains variables to store the *name* of the item, its *weight* and the *type* of item it is.

An item of exquisite creation and perfect manufacture is said to be a masterwork. As well as incurring an additional cost to buy, it grants extra bonuses for armor, and damage for weapons. For a weapon or armor to inherit magical properties, the items must also be a masterwork, as such a boolean value stored in the item class to lay claim to whether the item is a *masterwork* or not.

The cost to purchase an item is stored as both an integer *price* and a character string *currency*. This is due to the fact that an item may cost 1 gold or 1 silver piece. The integer allows for easy calculations, while the character string allows to identify the unit of currency that is to be calculated.

As before a constructor and destructor are present to handle dynamic memory, and a pair of functions exist to both give and return the value associated with each private data member.

Item subclasses

```
class armor : public item
{
   private:
      int armor_bonus;
      int max_dexerity;
      int armor_check_penalty;
      int arcane_failure;
      int speed_20;
      int speed_30;
};
```

On top of the properties inherited by the item class, the *armor* class contains a few additional data members. The *armor_bonus* represents the amount of protection the armor provides as a simple integer. Due to the way armor limits the movements of the body, *armor* carries with it a *max_dexterity* bonus, which indicates the highest dexterity bonus a character can receive because the armor prevents the character from gaining a higher dexterity bonus. Armor also affects a characters ability to perform some skills and as such a *armor_check_penalty* applies to some skills. The hindrance of some armor on hand movements can cause spell-casters to fail to cast a spell. This value is indicated by the *arcane_failure*. The *speed_20* and *speed_30* values represent the speed a character can move at while wearing the armor. Some characters can travel 30ft while other only 20ft, hence the two different values.

```
class weapon : public item
{
   private:
      int rank;
      int handed;
      int range;
      char* critical;
      char* damage;
      char* type;
};
```

Weapons can be categorised as a number of different “ranks” of weapons. Simple weapons can be used by anyone, while martial weapons and exotic weapons can only be used by someone that is trained to use them. The *rank* is expressed as an integer so-as to enable modification due to race. Dwarves and Elves have a natural ability to use particular weapons which to others are martial or exotic.

The *handed* value is to indicate whether a weapon is to be used with one or two hands. Expressing as an integer is again a way to allow calculations and changes. The *range* of a weapon is rather obvious though it only applies to ranged weapons. All weapons have an associated *critical* hit and *damage* rolls. They are expressed as char* because there are no calculations that need to be performed on them. The value on the character sheet is directly read from these values. The *type* of weapon describes whether it is a “slashing”, “bludgeoning”, “piercing” weapon or any combination of these.

```
struct item_spell
{
   char* spellName
   int charges;
};
```

The *item_spell* structure is for use in conjunction with the *magical_item* class, as some magical items can perform spells. This structure is unlike the *spell* class, in that this structure simply stores the spell name, and the number of *charges* (times a spell can be used). For a more descriptive reference to a spell, the *spell* class can be used to determine the extended properties of the spell in question.

```
class magical_item : public item
{
   private:
      char* feats[];
      char* notes[];

      node skills[];
      node abilities[];
      node combat_stats[];

      item_spell spell_list[];

   public:
      .
      .
      .
      void apply_modifiers();
};
```

Magical items are complex because they can affect almost anything about the character. As such a series of arrays are included in the *magical_item* class to separate an items effect into different sections. The effect to feats is stored in the list *feats*, skills in *skills*, abilities in *abilities*, combat statistics in *combat_stats* and additional notes in *notes*.

If a magical item can perform a spell, then the list of spells is stored in *spell_list* as the above mentioned *item_spell* structure.

As well as the usual *set_* and *get_* functions, a function called *apply_modifiers* exists which when called applies all of the modifiers listed in the various lists from this class. In this way all the modifiers are applied at once and there is minimal chance of applying the same modifier more than once.

Feat class

The *feat* class is used to apply any bonuses (modifiers) that are received for having a particular feat.

```
class feat
{
   private:
      char* name;
      node abilities[];
      node skills[];
      node combatStats[];
      node requirements[];
      char* notes;

   public:
      .
      .
      .
      void apply_modifiers();
};
```

Much like the *magical_item* class explained above, the *feat* class stores a list of modifiers, separated into different sections dependent on what they affect. They are all applied at once using the function *apply_modifiers* which steps through each list performing each modification until it reaches the end of all of the lists.

Spell class

There are multiple classes in Dungeons and Dragons, with a very large amount and variety of spells available to choose from. Some spells can be performed by members of different classes, and some are only performable by a particular class and even particular schooling under that class. The *spell* class is designed to hold the important spell-related data together.

```
class spell
{
   private:
      char* name;
      char* type;
      char* school;

      char_int_pair level;
};
```

The *name* represents the name of the spell. The *type* of spell indicates whether it is a Divine (cleric, paladin, ranger, and druid) or Arcane (sorcerer, wizard, and bard). The *school* specifies which schooling of magic the spell comes from, “illusion”, “abjuration”, “necromancy”, etc. As there are multiple classes that may be able to perform the spell, the *level* requirement is expressed as a *char_int_pair* so as to allow for different level requirements under different classes.

Race class

Particular values on a character sheet are affected depending on which *race* was chosen. The numbers that are affected have been collated into a single *race* class so as to provide greater control over them.

```
class race
{
   private:
      char* name;
      char* size;
      int base_speed;
      char* languages[];
      char* languages_choice[];
      char* fav_class;

      node attributes[];
      node proficiency[];
      node skill[];

      char* special[];
      char* dm_notes[];
};
```

The *name* of the race is specified under the variable of the same name. The *size* of the character is not expressed as a number but rather as “small” or “medium”. Special bonuses are awarded to the characters that are “small” and it is by *race* that a characters *size* is determined. The *base_speed* of a character is the speed with which they travel on foot, un-encumbered with items and without armor.

The *race* of a character also limits the number of *languages* that the character knows and can also the number of languages that the character can learn. These values are expressed as arrays of character strings in the variables *languages* and *languages_choice*.

The favoured class is a class with which a particular race has the most existing qualities and is the easiest class for a particular race to take up. The favoured class comes into play when a character wishes to train in multiple classes at once. If one of the classes the character takes up is the favoured class for their race, then that class is not counted towards any penalties. This is stored as a character string called *fav_class*.

The lists *attributes*, *proficiency*, and *skill* are all lists of potential bonuses or penalties associated with choosing that particular class. Much like previous classes, all these modifiers are applied at the same time using an *apply_modifiers* function.

The lists *special*, and *dm_notes* are similar to the last three lists in that they are changes to a character due to their chosen class, but *special* and *dm_notes* are simply notes to be appended on the character sheet, as there are no values with which to change for the particular bonus. These notes rely on the player and the dungeon master to utilise the bonuses or penalties during play.

Character-class class

A characters class can affect some of the basic values on a characters sheet. A player can also choose to train in multiple classes, which makes keeping track of the changes difficult. This class was developed with the intention of having one instance per different character-class the player should choose.

```
class char_class
{
   private:
      char* name;
      int level;

      int hit_die;
      int base_attack;
      int skill_points;

      bool save_fortitude;
      bool save_reflex;
      bool save_will;

      int spells_day;
      int spells_known;
      char* primary ability;

      char* special_feats[];
      char* class_skills[];
};
```

The name of the class is again represented as a character string variable called *name*. The current level of the character in this particular character-class is stored as an integer called *level*. The value of *hit_die* determines the maximum number of hit points a character can go up in this character-class. The *base_attack* stores the base attack bonus that a player receives due to their current character-class. The number of *skill_points* they have to use is stored as an integer.

The three boolean values for *save_fortitude*, *save_reflex*, and *save_will* determine whether the character-class has “poor” or “good” saving throws. Once that is determined the appropriate increase in fortitude, reflex and will saves can be calculated.

The *spells_day* and *spells_known* variables represent the number of spells a character can perform per day and the total number of spells they can learn is. This of course only applies to spell-casting classes (Bard, Cleric, Druid, Ranger, Sorcerer, and Wizard). These numbers are affected by which class a character is and by which level in that character-class a character is as well.

The *primary_ability* variable stores a character string, which contains the current class’ primary ability. A number of bonuses depend on how high the characters primary ability value is. For spell-casters this can mean extra spells per day or spells known.

The list *special_feats* contains all of the bonus feats that are awarded by training in the current character-class. These feats are added irrespective of how many feats the character already has.

The last list *class_skills* contains all of the names of the skills which are considered relative to the current character-class. Skills not on this list cost twice as many skill points to go up a rank in.

Though these are not all of the classes used in the program, these are the main classes that are related to the rules of Dungeons and Dragons.

Coding and naming conventions

Due to the large amount of people working on the project, we decided it important to create a coding and naming convention to ensure that all code developed for the project will have continuity, and allow for easier testing and collaboration of different segments of code. Following is a summary of some coding conventions that will be standard in all coding.

Block Comments

File Header:

```
/*********************************************************\
* Filename                                                *
* =========                                               *
* Project: D20 Character Generation                       *
* Created By:                                             *
* Last Edited By:                                         *
* Last Edited: 00/00/00                                   *
* Description:                                            *
*                                                         *
\*********************************************************/
```

Function Header:

```
/*********************************************************\
* function (parameters)                                   *
* parameter1 description:                                 *
* parameter2 description:                                 *
* ...                                                     *
* return value description:                               *
* function description:                                   *
\*********************************************************/
```

Identifiers

- multiple words separated by underscore (e.g. file_counter)
- non-constants use all lowercase
- constants are all uppercase
- identifiers must describe the function or value they perform/hold

Statement Braces

- curly braces appear on the line following the function name
- curly braces are inline with the function
- code begins one tab in from the curly brace on the next line
- switch cases appear one tab in, code to be performed appear another tab in

Identation

- Each statement block appears in a new indent

Variables

- variables defined at the beginning of the scope it is in
- variables declared in a scope inside a function should be commented with other variables for the function
- avoid global variables (except in extreme circumstances)

Editing Code

- Printing debugging information to cerr, should be highly commented to stand out
- Comment out the code to be changed, include a timestamp and the name of the person making the change, e.g.

```
// 13/03/04 Caleb
// while (1) {cin >> filename;}
for (int file_count=0;file_count<FILE_MAXFILES;file_count++)
    cin >> filename;
```

Code File Names

- Classes have there own ".cpp" and ".h" files
- File names are of the form `parent_subclass.(cpp,h.` (If a file is not part of a class, the filename must describe it's place in the program e.g. "ui_menus.h")

Pre-Compile Commands

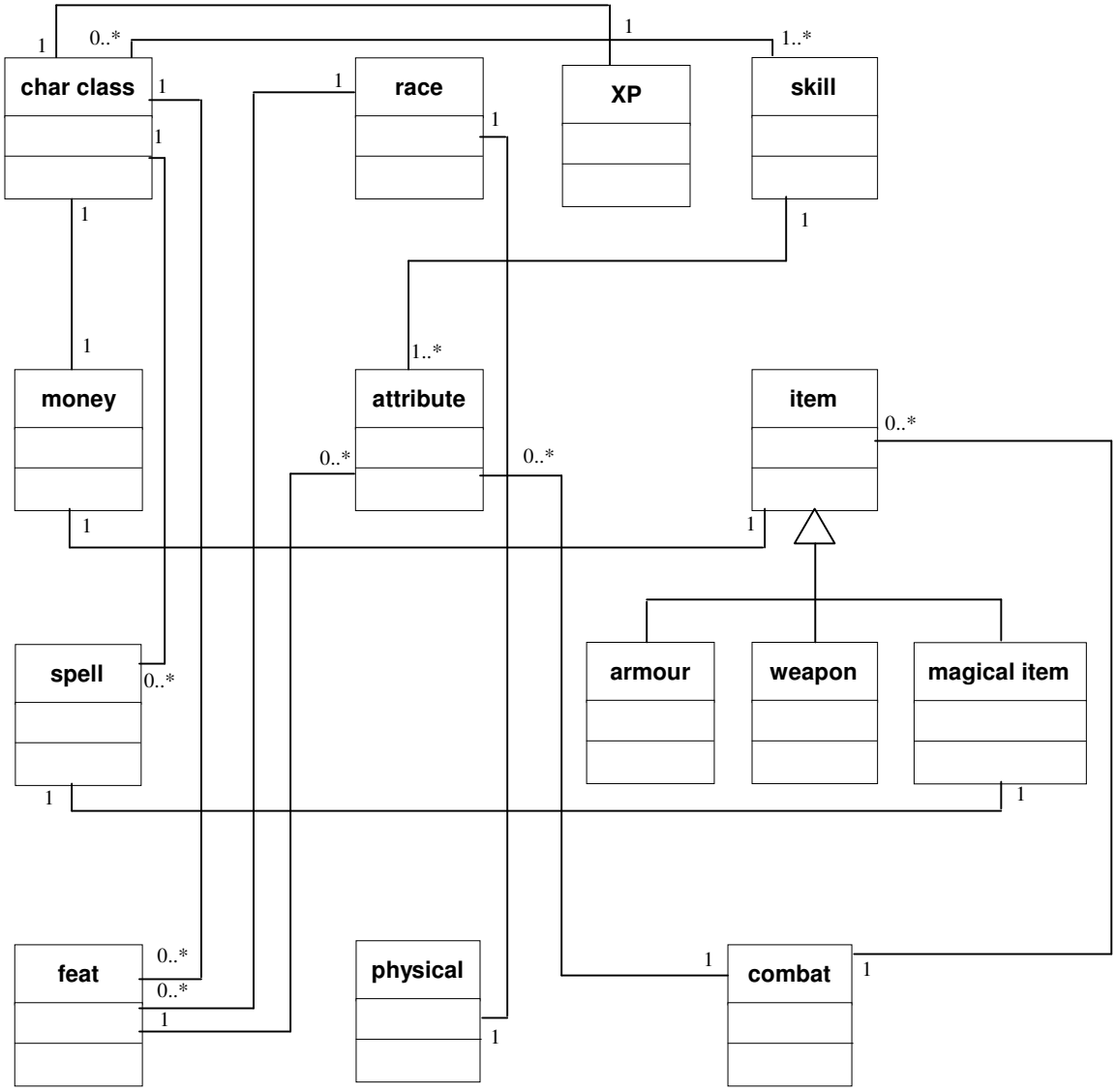
- Use of pre-compile commands in a header file to ensure it is not included more than once, e.g.

```
#ifndef HEADER_H
 .
 .
 // Code goes here
 .
 .
#define HEADER_H
#endif
```

- Use of pre-compile commands during debugging, e.g.

```
#define DEBUG
 .
 .
 .
#ifdef DEBUG
   cerr << this_value << endl;
#endif
```

Class diagram and relationships



Links in this diagram show relationships between the classes at a high level. The main purpose for this diagram is to allow easy identification of other classes that may be affected if class information is changed.

This class diagram focuses closely on the implementation of the main classes from the section above. Links in this diagram show relationships between the classes at a high level. The main purpose for this diagram is to allow easy identification of other classes that may be affected if class information is changed. *(For a complete specification of the system as a class diagram see index. This will be added in the next revision of the manual.)*

The Character Environment is a broad concept that encompasses the user interface and the shell.
Particular attention has to be paid to the *item* class as it has three sub-classes:

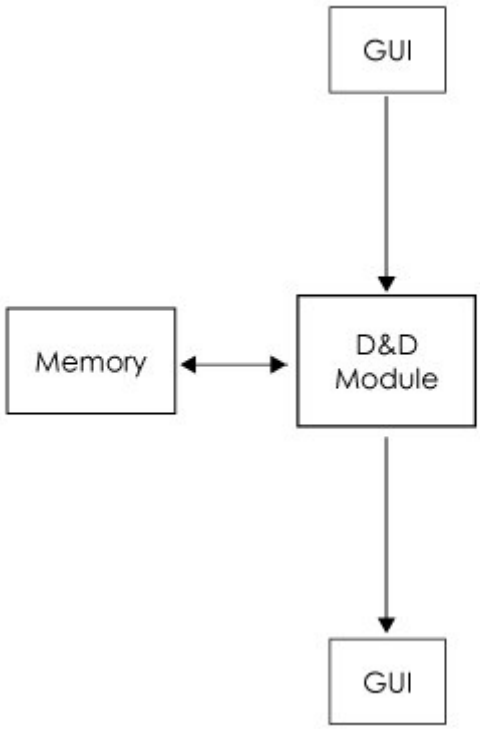
- Weapon
- Armour
- Magic item

The skill class' unary relationship is also of interest. Modification of a skill object can have an effect on other skill objects.

All other relationships are fairly straight-forward.

GUI interaction

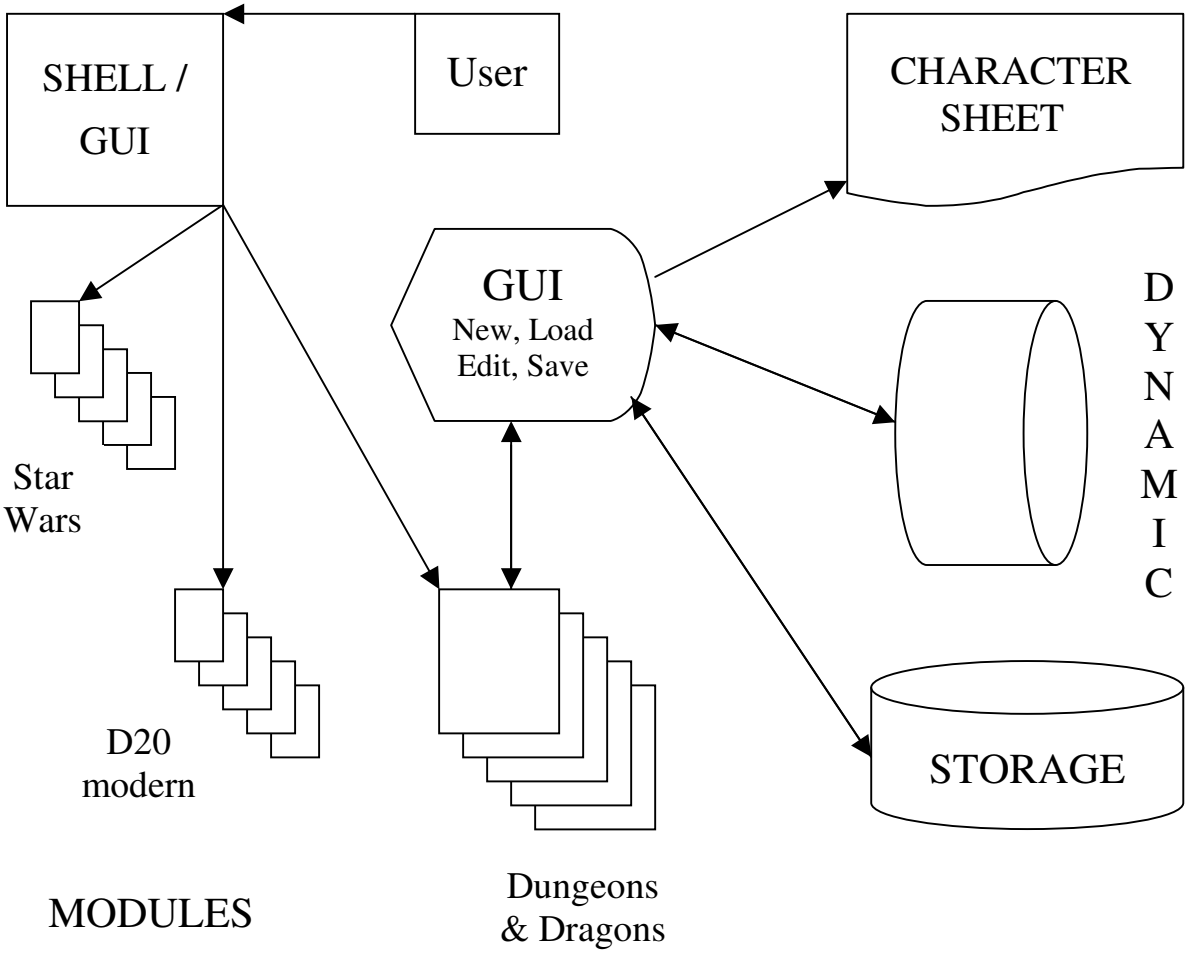
When the user interacts with the GUI, i.e. when they add their abilities scores into the system, the GUI sends signals to the D&D module functions that have slots listening for signals. When a slot hears a signal, the function in the D&D module then interacts with the memory setup for the program. It can either sent information, receive information or do both. Once the function has completed it's processing of data it then sends a signal back to the GUI, that in turns makes the appropriate changes to the data displayed to the user.



Shell and module design

The proposed solution for the implementation of the program will be to use a 'shell and module' design. This model will allow the encorporation many different d20 game rulesets for character generation and maintenance into the one program. From the shell, the user is able to access the various d20 rulesets, and for each create new, edit existing or create random characters.

A visualisation of the system is shown below.



The diagram shows the program processes go through when a user wants to use the Dungeons and Dragons module of the program.

- The user opens the program and enters the shell. They are then presented with a graphical user interface (GUI) dialogue, prompting them to select their d20 ruleset. In this case the Dungeons and Dragons module is selected.

- Once the appropriate ruleset is selected, the program then proceeds to leave the shell and enter the game module. All required classes and functions are loaded ready for use, and the appropriate GUI dialogue is displayed.

- From the displayed GUI, the user can then choose to create a new, edit an existing or create a random character. Information is constantly updated within the dynamic storage whilst modifications are taking place. The GUI controls all of the program functions – new, load, edit, save and creating a character sheet.
    - new: creates new dynamic storage,
    - load: retrieves information from hard storage,
    - edit: changes to information updated constantly in dynamic memory,
    - save: turns dynamic copy into a hard storage copy,
    - character sheet: takes information and places into a template.

- Once the user has finished with the selected module, they can go back to the program shell and select a different module, following the same process as shown above.

Important tables

Table of contents:

Table 3.1 – Base Save and Base Attack Bonuses

Class level	Base Save Bonus (Good)	Base Save Bonus (Poor)	Base Attack Bonus (Good)	Base Attack Bonus (Average)	Base Attack Bonus (Poor)
1st	+2	+0	+1	+0	+0
2nd	+3	+0	+2	+1	+1
3rd	+3	+1	+3	+2	+1
4th	+4	+1	+4	+3	+2
5th	+4	+1	+5	+3	+2
6th	+5	+2	+6/+1	+4	+3
7th	+5	+2	+7/+2	+5	+3
8th	+6	+2	8/+3	+6/+1	+4
9th	+6	+3	+9/+4	+6/+1	+4
10th	+7	+3	+10/+5	+7/+2	+5
11th	+7	+3	+11/+6/+1	+8/+3	+5
12th	+8	+4	+12/+7/+2	+9/+4	+6/+1
13th	+8	+4	+13/+8/+3	+9/+4	+6/+1
14th	+9	+4	+14/+9/+4	+10/+5	+7/+2
15th	+9	+5	+15/+10/+5	+11/+6/+1	+7/+2
16th	+10	+5	+16/+11/+6/+1	+12/+7/+2	+8/+3
17th	+10	+5	+17/+12/+7/+2	+12/+7/+2	+8/+3
18th	+11	+6	+18/+13/+8/+3	+13/+8/+3	+9/+4
19th	+11	+6	+19/+14/+9/+4	+14/+9/+4	+9/+4
20th	+12	+6	+20/+15/+10/+5	+15/+10/+5	+10/+5

Table 3.2 – Experience and level-dependent benefits

Ch'ter level	XP	Class Skill Max Ranks	Cross-class Skill Max Ranks	Feats	Ability Score Increases
1st	0	4	2	1st	–
2nd	1,000	5	2-1/2	–	–
3rd	3,000	6	3	2nd	–
4th	6,000	7	3-1/2	–	1st
5th	10,000	8	4	–	–
6th	15,000	9	4-1/2	3rd	–
7th	21,000	10	5	–	–
8th	28,000	11	5-1/2	–	2nd
9th	36,000	12	6	4th	–
10th	45,000	13	6-1/2	–	–
11th	55,000	14	7	–	–
12th	66,000	15	7-1/2	5th	3rd
13th	78,000	16	8	–	–
14th	91,000	17	8-1/2	–	–
15th	105,000	18	9	6th	–
16th	120,000	19	9-1/2	–	4th
17th	136,000	20	10	–	–
18th	153,000	21	10-1/2	7th	–
19th	171,000	22	11	–	–
20th	190,000	23	11-1/2	–	5th

Table 3.7 – Dieties

Deity	Alignment	Domains	Typical Worshipers
Heironeous, god of valor	Lawful good	Good, Law, War	Paladins, fighters, monks
Moradin, god of the dwarves	Lawful good	Earth, Good, Law, Protection	Dwarves
Yondalla, goddess of the halflings	Lawful good	Good, Law, Protection	Halflings
Ehlonna, goddess of the woodlands	Neutral good	Animal, Good, Plant, Sun	Elves, gnomes, half-elves, halflings, rangers, druids
Garl Glittergold, god of the gnomes	Neutral good	Good, Protection, Trickery	Gnomes
Pelor, god of the sun	Neutral good	Good, Healing, Strength, Sun	Rangers, bards
Corellon Larethian, god of the elves	Chaotic good	Chaos, Good, Protection, War	Elves, half-elves, bards
Kord, god of strength	Chaotic good	Chaos, Good, Luck, Strength	Fighters, barbarians, rogues, athletes
Wee Jas, goddess of death and magic	Lawful neutral	Death, Law, Magic	Wizards, necromancers, sorcerers
St. Cuthbert, god of retribution	Lawful neutral	Destruction, Law, Protection, Strength	Fighters, monks, soldiers
Boccob, god of magic	Neutral	Knowledge, Magic, Trickery	Wizards, sorcerers, sages
Fharlanghn, god of roads	Neutral	Luck, Protection, Travel	Bards, adventurers, mercharnts
Obad-Hai, god of nature	Neutral	Air, Animal, Earth, Fire, Plant, Water	Druids, barbarians, rangers
Olidammara, god of thieves	Chaotic neutral	Chaos, Luck, Trickery	Rogues, bards, thieves
Hextor, god of tyranny	Lawful evil	Destruction, Evil, Law, War	Evil fighters, monks
Nerull, god of death	Neutral evil	Death, Evil, Trickery	Evil necromancers, rogues
Vecna, god of secrets	Neutral evil	Evil, Knowledge, Magic	Evil wizards, sorcerers, rogues, spies
Erythnul, god of slaughter	Chaotic evil	Chaos, Evil, Trickery, War	Evil fighter, barbarians, rogues
Gruumsh, god of the orcs	Chaotic evil	Chaos, Evil, Strength, War	Half-orcs, orcs

Table 2.1 – Racial ability adjustments

Race	Ability Adjustment	Favoured Class
Human	None	Any
Dwarf	+2 Constitution, -2 Charisma	Fighter
Elf	+2 Dexterity, -2 Strength	Wizard
Gnome	+2 Constitution, -2 Strength	Bard
Half-elf	None	Any
Half-orc	+2 Strength, -2 Intelligence[1], -2 Charisma	Barbarian
Halfling	+2 Dexterity, -2 Strength	Rogue

1 A half-orc's starting Intelligence score is always at least 3. If this adjustment would lower the character's score to 1 or 2, his score is nevertheless 3.

Table 6.4 – Random starting ages

Race	Adulthood	Barbarian Rogue Sorcerer	Bard Fighter Paladin Ranger	Cleric Druid Monk Wizard
Human	15 years	+1d4	+1d6	+2d6
Dwarf	40 years	+3d6	+5d6	+7d6
Elf	110 years	+4d6	+6d6	+10d6
Gnome	40 years	+4d6	+6d6	+9d6
Half-elf	20 years	+1d6	+1d6	+2d6
Half-orc	14 years	+1d4	+1d6	+2d6
Halfling	20 years	+2d4	+3d6	+4d6

Table 6.6 – Random height and weight association

Race	Base Height	Height modifier	Base weight	Weight modifier
Human, male	4’ 10”	+2d10	120 lb.	X (2d4) lb.
Human, female	4’ 5”	+2d10	85 lb.	X (2d4) lb.
Dwarf, male	3’ 9”	+2d4	130 lb.	X (2d6) lb.
Dwarf, female	3’ 7”	+2d4	100 lb.	X (2d6) lb.
Elf, male	4’ 5”	+2d6	85 lb.	X (1d6) lb.
Elf, female	4’ 5”	+2d6	80 lb.	X (1d6) lb.
Gnome, male	3’ 0”	+2d4	40 lb.	X 1 lb.
Gnome, female	2’ 10”	+2d4	35 lb.	X 1 lb.
Half-elf, male	4‘ 7”	+2d8	100 lb.	X (2d4) lb.
Half-elf, female	4’ 5”	+2d8	80 lb.	X (2d4) lb.
Half-orc, male	4’ 10”	+2d12	150 lb.	X (2d6) lb.
Half-orc, female	4’ 5”	+2d12	110 lb.	X (2d6) lb.
Halfling, male	2’ 8”	+2d4	30 lb.	X 1 lb.
Halfling, female	2’ 6”	+2d4	25 lb.	X 1 lb.

Table 2: Hit die / class

HD Type	Class
d4	Sorcerer, wizard
d6	Bard, rogue
d8	Cleric, druid, monk, ranger
d10	Fighter, paladin
d12	Barbarian

Table 7.4 – Tiny & large weapon damage

Medium weapon damage	Tiny weapon damage	Large weapon damage
1d2	–	1d3
1d3	1	1d4
1d4	1d2	1d6
1d6	1d3	1d8
1d8	1d4	2d6
1d10	1d6	2d8
1d12	1d8	3d6
2d4	1d4	2d6
2d6	1d8	3d6
2d8	1d10	3d8
2d10	2d6	4d8

Table 7.3 – Trade goods

Cost	Item
1cp	One pound of wheat
2cp	One pound of flour, or one chicken
1sp	One pound of iron
5sp	One pound of tobacco or copper
1gp	One pound of cinnamon, or one goat
2gp	One pound of giner or pepper, or one sheep
3gp	One pig
4gp	One square yard of linen
5gp	One pound of salt or silver
10gp	One square yard of silk, or one cow
15gp	One pound of saffron, or cloves, or one ox
50gp	One pound of gold
500gp	One pound of platinum

Table 4-1: Skill Points Per Level

Class	1st-Level Skill Points[1]	Higher-Level Skill Points[2]
Barbarian	(4 + Int modifier) * 4	4 + Int modifier
Bard	(6 + Int modifier) * 4	6 + Int modifier
Cleric	(2 + Int modifier) * 4	2 + Int modifier
Druid	(4 + Int modifier) * 4	4 + Int modifier
Fighter	(2 + Int modifier) * 4	2 + Int modifier
Monk	(4 + Int modifier) * 4	4 + Int modifier
Paladin	(2 + Int modifier) * 4	2 + Int modifier
Ranger	(6 + Int modifier) * 4	6 + Int modifier
Rogue	(8 + Int modifier) * 4	8 + Int modifier
Sorcerer	(2 + Int modifier) * 4	2 + Int modifier
Wizard	(2 + Int modifier) * 4	2 + Int modifier
1 **Humans** add +4 to this total at 1st-Level.		
2 **Humans** add +1 to each new level.		

Table 7-1: Random Starting Gold

Class	Amount (average)	Class	Amount (average)
Barbarian	4d4 * 10 (100gp)	Paladin	6d4 * 10 (150gp)
Bard	4d4 * 10 (100gp)	Ranger	6d4 * 10 (150gp)
Cleric	5d4 * 10 (125gp)	Rogue	5d4 * 10 (125gp)
Druid	2d4 * 10 (50gp)	Sorcerer	3d4 * 10 (75gp)
Fighter	6d4 * 10 (150gp)	Wizard	3d4 * 10 (75gp)
Monk	5d4 (12gp, 5sp)		

Table 4-2: Skills

Skill	Bbn	Brd	Clr	Drd	Ftr	Mnk	Pal	Rgr	Rog	Sor	Wiz	Untrained	Key Ability
Appraise	cc	C	cc	cc	cc	cc	cc	cc	C	cc	cc	Yes	Int
Balance	cc	C	cc	cc	cc	C	cc	cc	C	cc	cc	Yes	Dex1
Bluff	cc	C	cc	cc	cc	cc	cc	cc	C	C	cc	Yes	Cha
Climb	C	C	cc	cc	C	C	cc	C	C	cc	cc	Yes	Str1
Concentration	cc	C	C	C	cc	C	C	C	cc	C	C	Yes	Con
Craft	C	C	C	C	C	C	C	C	C	C	C	Yes	Int
Decipher Script	cc	C	cc	cc	cc	cc	cc	cc	C	cc	C	No	Int
Diplomacy	cc	C	C	C	cc	C	C	cc	C	cc	cc	Yes	Cha
Disable Device	cc	cc	cc	cc	cc	cc	cc	cc	C	cc	cc	No	Int
Disguise	cc	C	cc	cc	cc	cc	cc	cc	C	cc	cc	Yes	Cha
Escape Artist	cc	C	cc	cc	cc	C	cc	cc	C	cc	cc	Yes	Dex1
Forgery	cc	cc	cc	cc	cc	cc	cc	cc	C	cc	cc	Yes	Int
Gather Information	cc	C	cc	cc	cc	cc	cc	cc	C	cc	cc	Yes	Cha
Handle Animal	C	cc	cc	C	C	cc	C	C	cc	cc	cc	No	Cha
Heal	cc	cc	C	C	cc	cc	C	C	cc	cc	cc	Yes	Wis
Hide	cc	C	cc	cc	cc	C	cc	C	C	cc	cc	Yes	Dex1
Intimidate	C	cc	cc	cc	C	cc	cc	cc	C	cc	cc	Yes	Cha
Jump	C	C	cc	cc	C	C	cc	C	C	cc	cc	Yes	Str1
Knowledge (arcana)	cc	C	C	cc	cc	C	cc	cc	cc	C	C	No	Int
Knowledge (architecture and engineering)	cc	C	cc	cc	cc	cc	cc	cc	cc	cc	C	No	Int
Knowledge (dungeoneering)	cc	C	cc	cc	cc	cc	cc	C	cc	cc	C	No	Int
Knowledge (geography)	cc	C	cc	cc	cc	cc	cc	C	cc	cc	C	No	Int
Knowledge (history)	cc	C	C	cc	cc	cc	cc	cc	cc	cc	C	No	Int
Knowledge (local)	cc	C	cc	cc	cc	cc	cc	cc	C	cc	C	No	Int
Knowledge (nature)	cc	C	cc	C	ccc	cc	cc	C	cc	cc	C	No	Int
Knowledge (nobility and royalty)	cc	C	cc	cc	cc	cc	C	cc	cc	cc	C	No	Int
Knowledge (religion)	cc	C	C	cc	cc	C	C	cc	cc	cc	C	No	Int
Knowledge (the planes)	cc	C	C	cc	cc	cc	cc	cc	cc	cc	C	No	Int
Listen	C	C	cc	C	cc	C	cc	C	C	cc	cc	Yes	Wis
Move Silently	cc	C	cc	cc	cc	C	cc	C	C	cc	cc	Yes	Dex1
Open Lock	cc	cc	cc	cc	cc	cc	cc	cc	C	cc	cc	No	Dex
Perform	cc	C	cc	cc	cc	C	cc	cc	C	cc	cc	Yes	Cha
Profession	cc	C	C	C	cc	C	C	C	C	C	C	No	Wis
Ride	C	cc	cc	C	C	cc	C	C	cc	cc	cc	Yes	Dex
Search	cc	cc	cc	cc	cc	cc	cc	C	C	cc	cc	Yes	Int
Sense Motive	cc	C	cc	cc	cc	C	C	cc	C	cc	cc	Yes	Wis
Sleight of Hand	cc	C	cc	cc	cc	cc	cc	cc	C	cc	cc	No	Dex1
Speak Language	cc	C	cc	cc	cc	cc	cc	cc	cc	cc	cc	No	None
Spellcraft	cc	C	C	C	cc	cc	cc	cc	cc	C	C	No	Int
Spot	cc	cc	cc	C	cc	C	cc	C	C	cc	cc	Yes	Wis
Survival	C	cc	cc	C	cc	cc	cc	C	cc	cc	cc	Yes	Wis
Swim	C	C	cc	C	C	C	cc	C	C	cc	cc	Yes	Str2
Tumble	cc	C	cc	cc	cc	C	cc	cc	C	cc	cc	No	Dex1
Use Magic Device	cc	C	cc	cc	cc	cc	cc	cc	C	cc	cc	No	Cha
Use Rope	cc	cc	cc	cc	cc	cc	cc	C	C	cc	cc	Yes	Dex

1 Armor check penalty applies to checks.

2 Double the normal armor check penalty applies to checks.

CSCI321 Project Diary
Week ending 13th March 2004 – Week 4

January

- *Mid January* – project group formed
  - Caleb Avery cda96
  - Mark Boxall mcb36
  - Mark Hellmund mkh97
  - Mavis Shaw mrs26
  - Michael Tonini mt02

- *Late January* – informal project meeting to discuss the possibility of proposing a project to a lecturer for completion over the course of CSCI321. The idea of creating a character generation and maintenance program for the popular role-playing game Dungeons and Dragons was decided upon, with a proposal to be drawn up within the next week for presentation to Computer Science lecturer Peter Castle.

February

- *Early February* – Initial proposal for the selected project completed, and presented to Peter Castle by Mark Boxall. The lecturer accepted the project proposal and offered to supervise the project (see attached proposal).

- *Mid February* – Group members begin work on the project through familiarisation with the game Dungeons and Dragons (D&D). All members obtained a copy of the D&D 3.5 ruleset, and created individual characters via the creation tool 'E-tools', then through traditional character sheets.

March

Week 2

- *4th March, first formal meeting* – Full attendance; Project roles specified – these are a general guideline to the main roles of each person within the development team, however all members are expected to participate in all aspects of the project lifecycle. The roles are based on an IT development team situation, and are as follows:

  - **Mark Boxall** – *Project leader & Interface design*
    Monitor and manage all aspects of system design and development, including project timelines and required resources.
  - **Michael Tonini** – *Business/requirements analyst*
    Acquires comprehensive understanding of the business, along with their requirements and needs for the new software system.
  - **Mavis Shaw** – *Systems analyst*
    Designs a new system based on the users current system requirements derived from the clients needs.
  - **Caleb Avery & Mark Hellmund** – *Systems developers*
    Using design specifications, create using a set of computer programming languages a system to meet user requirements.

- Following on from the meeting, the following points were discussed:
    - Developer environment *QT* (C++ language compiler) to be used for project development,
    - System design day to be organised as soon as project goals and system specifications have been determined,
    - Mark B. to create an online diary system to house individual diaries (however each member is expected to keep their own), and an area to store group documentation and a message board for communications,
    - Caleb and Mark H. to develop a Coding and Naming Specification for all project coding to be based on, to ensure continuity of all code,
    - Goals and submission dates discussed, first assessable task due in week 4 (project diary).

- *7th March, D&D campaign* – All members and generated characters set out on their first campaign to gain some experience in the gameplay of D&D. After successfully completing an expedition filled with many humanoids and monstrous spiders, the party containing a druid (Michael), a ranger (Mavis), a rogue (Mark H.) and sourceress (Caleb), each gained enough experience points to move up to level 2, allowing us to see how this process is completed.

Week 3

- *12th March, formal meeting* – Full attendance, including Peter; discussed the main goals for the project throughout the course of this year. The finished product aims to fulfil these three main requirements of the system:

    - Character generation,
    - Character maintenance,
    - Random character generation (time-dependant).

  The following are issues the program needs to address or wants to accomplish:

    - Program will be designed to allow for rule changes to be made as easy and simple as possible,
    - Provide a report on changed character statistics when they 'level up',
    - A log system that keeps track of all statistics that change throughout the life of the character,
    - Help systems that describe where certain values or modifiers come from if user is unsure (ie. roll-over pop-up),
    - Functionality that allows users to add their own custom special items that the program will recognise as proper items,
    - Ensure that the character sheets generated by the program remain as close as possible to the current D&D character sheets,
    - *'Quick sheet'* containing useful values for quick reference.

- Current progress:
    - Mark B. has created a project website as discussed last week, containing facilities for diary entries, message boards and document storage. The temporary address until we are allocated university webspace is http://www.goldfishgifts.com.au/csci321/.

- Mavis has begun project design through identifying the necessary areas that need to be covered within the character generation section. From requirements determined from this meeting, project design can now properly begin.
- Michael has been taking minutes at each meeting, and distributing them to group members via email. Minutes will soon be available from the website. Michael is also inquiring into when we will be receiving university webspace to store all of our project material.

Week 4

- *16th March, meeting* – Full attendance; discussion on design issues in more detail. The proposed D&D program has been discussed, and a concept diagram has been created to symbolise the different aspects of character development, which will allow easier identification of program modulation at a later stage.

- Current progress:
  - Mavis has collated all information to be contained within the group project diary to be submitted at the end of week 4,
  - Michael has contacted Koren in regards to project webspace; will be available to all groups as of week 5,
  - Current temporary website will be ported to university webspace once it becomes available, including the message board, document storage and individual diaries,
  - Further clarification of user manual discussed, contact with Peter will be made for more information.

- The group split up into two groups to discuss both design techniques and coding specifications, which produced the following outcomes:
  - Caleb and Mark H. have drawn up a Coding and Naming Specification for all project coding to be based upon. These guidelines will help to make all code uniform, allowing easier modifications and testing later on to code written by anyone in the group.
  - Mark B, Mavis and Michael discussed design issues, producing an overall conceptual design model of the current system, which breaks the problem down into smaller parts. Following on from this, the races and classes section are the first to be analysed, with all group members being allocated a set to note all modifiers and any relevant information about each race/class.

	Caleb	**Mark B.**	**Mark H.**	**Mavis**	**Michael**
Races	- Human - Dwarf	- Half-elves	- Halfling	- Gnome	- Elf - Half-orc
Classes	- Barbarian - Sourcerer	- Cleric - Paladin - Wizard	- Monk - Rogue	- Fighter - Ranger	- Bard - Druid

Race information to be completed by next meeting, classes by next week.

- Meeting with Peter Castle set to Thursdays 11:00am – 12:00pm weekly.

- 18th March, meeting – Mark B, Mavis, Michael, Peter; Reviewed with Peter our current situation, and reviewed work done on races. Mark B has organised a structure that visually presents all information about each race in a quick to understand format, and hopefully can be applied to each section of the design process.
- Current progress:
    - Caleb and Mark H. completed the Naming and Coding Specification, although some suggested changes will need to be made. Will be reviewed at next meeting,
    - Mavis will complete project diary and hand to Peter on Friday. Also she is compiling information on skills, in particular any statistics in relation to synergy bonuses,
    - Mark B. to start compiling information on feats,
    - Caleb and Michael to start compiling information on spells.
- Next meeting – Tuesday 23rd March, 6:30pm

CSCI321 Project Diary –' d20'
Week ending 23rd April 2004 – Week 8

Week 5 (22nd – 28th March)

- This week began discussions on how aspects of the D&D ruleset will modify each other. Information on the races and classes has been collaborated, and all of the bonuses and modifiers to other areas of the game identified (ie. racial bonuses). Within the feats section of the game we discovered that some skills can modify certain character skills and attributes through their synergies, all of these have now been extracted and put in a table for future use.
- Created some mindmap diagrams to visualise what attributes each 'object' owns. These were created for both race and class, and for items – including the subclasses of weapons, armour and magical items. These will be displayed within the technical manual.
- Began talks about the preliminary user manual due in week 7. Main focus was on how the program would flow, with ideas thrown around and left to think about until next week. No installation guide will be required for the preliminary user manual.

Week 6 (29th March – 4th April)

- Main discussion about preliminary user manual was undertaken at Tuesdays meeting, with program processes being defined to allow for the program to be broken down into its specific areas. The program was broken up into 3 main sections; create a new character, edit an existing character and create a random character. Most of the meeting was spent talking about the graphical user interface design for each section, mainly for the create/edit character section. Mark B. took all of our ideas away to begin working on the screen shots using the development tool QT.
- For creating a new, edititing or creating a random character processes, we discussed exactly what would occur for each instance. From this, Mavis went away to create some data flow diagrams to assist everyone in quickly viewing the processes that occur for each section of the program.

Week 7 (5th – 11th April)

- User manual screen shots were all completed this week, allowing Michael and Mavis to start writing the content. The total number of GUI screen shots was 14, covering every section of the proposed final program. The final document will contain an introduction, explanation of all GUI components, FAQ's, and glossary/appendices/index. User manual was to be completed at the end of this week, but ran over into Easter recess.
- Discussed implementation, leading up to what will actually appear within the preliminary technical manual, due in week 9. Due to the programs ultimate goal of catering for more than just the D&D ruleset, the idea of creating a 'shell' to house the main program functions was suggested, with each ruleset being treated as an add-on module to the program. The group was unsure as to whether this would be suitable, so further clarification as to the program specifications would have to be made with Peter.

Easter Recess (12th – 18th April)

- User manual completed, with full explanations of all GUI screen shots as well as general information about the program, the project group, and also a glossary and index. Final draft was 23 pages long, Mavis delivering to Peter later in the week.
- No meetings this week due to Easter break and everyone wanting some well earned time off!

Week 8 (19th – 25th April)

- The group realises that we are slightly behind in the implementation design of the program for the technical manual, so discussions began on how we are going to create the program. QT will be used to create the GUI interface, with supporting C++ code interacting with it in the background. The idea of creating a supporting shell for the program has been proposed, which will load in information from text files based on the d20 system that the user chooses. Further investigation must be done on this idea to see if it will be feasable and efficient.
- Extra meeting was held on Sunday to make further progress on implementation ideas. The group has come up with 15 base classes that will store character information specifically for the D&D 3.5 rule set. The next step will be determining relationships between all of the different classes.
- Next deliverables due include project diary due this week (week 8), and the preliminary technical user manual, due next week (week 9).

Minutes Of Meeting CSCI321 – 4th March, 2004

Attendance:

Mark Hellmund	Caleb Avery
Mark Boxall	Mavis Shaw
Michael Tonini	

Designated roles

Mavis proposed/team carried: While the team will be working together on every aspect of the project these designated roles provide an area for which each team member is responsible:

- Caleb - Programmer (C++/Software)
- Mark H - Programmer (C++/Hardware)
- Mavis - Software Analyst
- Michael - Business (Requirements) Analyst
- Mark B - Interface Design (Team Leader)

Other Points

- Quick overview of the proposed development environment (QT3)

- A point was made that this software will not need to use a database, as the overhead will be too costly. (Documented for reference in design stage.)

- Proposed planning day with butcher paper to work on understanding and design of the project.

Current tasks

- Mark B will have an interactive diary online within the next week to store minutes and diary entries online AS A BACKUP (Please not that it is important to also keep your own copy.)

- Caleb and Mark H will develop a Coding and Naming Standard specification that they expect to have done in the next week.

Goals for the semester

Week 4 - Diary submission
Week 7 - Preliminary User Manual submission
Week 8 - Diary submission
Week 9 - Preliminary Technical Design Manual submission
Week 12 - Diary submission
- Project Progress Website

Next Meeting

Time: Sunday 7th March 2004 1:30PM
Place: Mark Boxall's

Minutes Of Meeting CSCI321 – 16th March, 2004

Attendance:

Mark Hellmund	Caleb Avery
Mark Boxall	Mavis Shaw
Michael Tonini	

Discussion

- Review of project diary to be submitted this week.
- Organised meeting time of 11:30-12:30 Thursday with Peter. Needs to be confirmed.
- Michael contacted Koren about university storage. Details will be given week 5.
- Current temporary website will be ported to university storage when it is available.
- Point raised that we need to clarify user manual expectations.
- The group split up. Caleb and Mark H reviewed coding specifications.
- Mark B, Mavis and Michael drew up a context diagram of the overall D&D System.

Progress

- Mavis compiled the project diary so far.

Current tasks

- Mark B plans to have the website fully operational within two weeks.
- Everyone given races and classes to study and model:

	Mark B	Mark H	McCaleb	Mavis	Michael
Race:	Half Elves	Halfling	Human	Gnome	Elf
			Dwarf		Half-Orc
Class:	Cleric	Monk	Sorcerer	Fighter	Bard
	Paladin	Rogue	Barbarian	Ranger	Druid
	Wizard				

- Races to be finished Tuesday next week and Classes Thursday at the very latest.
- Mark B sending around an example of the mapping diagram of specific rules.

Next Meeting

Time: Thursday 18th March 11:30am –12:30pm
Place: Building 3

Minutes Of Meeting CSCI321 – 18th March, 2004

Attendance:

Mark Boxall	Caleb Avery
Michael Tonini	Peter Castle
Mavis Shaw	

Discussion

- Mavis put forward the submission of the project diary.
- User manual details clarified with Peter.
- Code Specification presented to Peter. The suggestion was made to print debug code to cerr and maybe have a debug flag also.
- Item creation raised as a complex issue. Yet to be looked at.

Progress

- Everyone has commenced their modelling tasks.

Current tasks

- Caleb to send coding guidelines to the group.
- Mavis modelling skills section. Mavis will also be submitting the project diary on Friday.
- Mark Boxall doing Feats.
- Caleb and Michael doing spells.

Next Meeting

Time: Tuesday 23rd March 6:30pm
Place: TBA

Minutes Of Meeting CSCI321 – 22nd March, 2004

Attendance:

Mark Boxall	Caleb Avery
Michael Tonini	Mark Hellmund
Mavis Shaw	

Discussion

- Mavis dropped the project Diary off on Friday.
- Height and weight attributes raised (by Caleb) Decision made they are general attributes not linked to any other part of the rules.
- Caleb mentioned an addition to coding spec. Filenames and class procedures. (to be reviewed.
- Mark B suggested the possibility of the openness this program will need in allowing a greater rule set.
- *Attributes* will need to be defined in the user manual.
- Michael pointed out that there are two and a half weeks until the preliminary user manual is due.
- The program is to contain in game Item modifications and item modification external to the game.

Progress

- Everyone completed previous weeks tasks.
- Tonight all race maps were compiled to look for similarities and produce a more genetic structure for this set of rules.

Current tasks

- Mavis is putting all the data brought forward in the meeting into tables.
- Mavis and Michael will work together on a user manual layout.
- Mark B to commence looking at overall rules.
- Caleb revising code specification. (There are some errors.)
- Unassigned task, to start modelling ITEMS!!

Next Meeting

Time: Thursday 25th March 11:00am
Place: Building 3

Minutes Of Meeting CSCI321 – 25th March, 2004

Attendance:

Mark Boxall	Caleb Avery
Mavis Shaw	Mark Hellmund
Peter Castle	

Apology: Michael Tonini started his work-integrated scholarship today and so could not attend the meeting.

Discussion

- No installation guide will be needed for the Preliminary User manual.
- Discussion on how house rules will be implemented.
- Peter will be providing the group with the current spreadsheets that he uses to assist in brainstorming our solution.
- Quick sheet…
- Diagrams for item, weapons, armour, magic items

Progress

- Races and skills synergy summaries completed.

Current tasks

- Overall rules and classes to be completed.
- Development on GUI has to be started.
- User Manual for week 7:
- Introduction to program/Install
- Requirements for character generation utility.
- GUI guide
- FAQ
- Index, Glossary, contents, appendix

Next Meeting

Time: Tuesday 30th March 7:00pm

Place: Mavis' house

Minutes Of Meeting CSCI321 – 30th March, 2004

Attendance:

Mark Boxall	Caleb Avery
Mavis Shaw	Mark Hellmund
Michael Tonini	

Discussion

- Brainstorm session on storage methods.
- Large discussion on how items will be factored into the program being that they can modify anything within the game.

Current tasks

- Mark is continuing to work on screen shots for the program so that the user preliminary user manual can be done

Minutes Of Meeting CSCI321 – 6th April, 2004

Attendance:

Mark Boxall	Caleb Avery
Mavis Shaw	Mark Hellmund
Michael Tonini	

Discussion

- Mavis showed the progress on the preliminary user manual.
- Suggestion that a shell may be needed to implement our program correctly.
- Large discussion on whether a shell is the correct solution or not.
- The group reviewed the layout of some of the GUI components that Mark has been designing.
- Caleb, Mavis and Michael worked through the functioning of the system so that DFDs can be drawn.

Progress

- Mavis and Michael will be looking at items for the Technical manual.
- Mark B has been working on the GUI.

Current tasks

- Every one has to start thinking about program functionality in order to complete the tech manual.
- Development on GUI.
- User Manual for this week:
- Introduction to program
- Requirements for character generation utility.
- GUI guide
- FAQ
- Index, Glossary, contents, appendix

Minutes Of Meeting CSCI321 – 20th April 2004

Attendance:

Caleb Avery	Mavis Shaw
Mark Boxall	Michael Tonini
Mark Hellmund	

Points of Progress

- Mavis presented a hard copy of the User manual that she and Mark B have been working on. It is open for criticism and addition.
- Mavis also presented DFDs that she had been working on from previous discussions with Caleb and Mark H.
- Before the meeting Mavis and Mark H had been discussing implementation issues.
- Caleb has done some brainstorming on implementation.

General Discussion

- There was discussion and clarification of what exactly QT3 will be doing within our project. Mark B made the point that QT3 is just a library.
- Discussion about implementation with particular attention paid to skill as a start.
- Multiple files for character role back was suggested.
- We really have to get stuck into the technical side of the project.

Current Tasks

- Everyone is brainstorming for our LONG weekend sessions.
- Mark H and Caleb are going to ask Peter some questions.
- Mark B is making a diagram of the core rules.

Next Meeting:
Sunday- Monday (if we need it)
Location TBA.

Minutes Of Meeting CSCI321 – 27th April 2004

Attendance:

Caleb Avery	Mavis Shaw
Mark Boxall	Michael Tonini
Mark Hellmund	

Points of Progress

- Mavis and Caleb presented the classes storage structures.

General Discussion

- Mavis presented her idea of a shell implementation:
- There was a general brainstorm on shell implementation.
  - .dll files

- Action: - Implement basics to the program.
  - Add extras (eg. Shell).

- The group reviewed the storage structures of the classes.
- The point was made that most structures need to be stored in arrays so that they can be expanded.
- Spell clarification- The group investigated and clarified the table for number of spells within the game and how it worked.

Current Tasks

- Michael to do an order of processes flowchart.
- Type in important D&D tables. (pg. 22,23,123,114,162)
- Technical manual layout.
- Mark B to organize the project web move to uni account.

Next Meeting:

Meeting: 4th May Tuesday