UNIVERSITY OF HERTFORDSHIRE
School of Computer Science

Modular BSc Honours in Computer Science

6COM0282 – Computer Science Project

Procedural Content Generation of Dungeon Environments

Author's signature

_____

Supervised by: Dr. Ian Bradford

Abstract

Traditionally, content for multimedia applications such as video games has been produced manually, but recent advances in technology are making it infeasible to continue producing the required amount of content by hand. Procedural Content Generation is a specialist application of Computer Science that has the potential to resolve this issue by partially automating the production of this content, but this technique has yet to enter widespread use. An identified reason for this is that the majority of existing generators are bespoke, intended for use by just one specific software program.

This report documents the creation of a fully generic, object-oriented generator of “dungeon” levels, which are defined in the video games industry as environments consisting of rooms interconnected by corridors. This includes the high-level design of the generator, the practical task of implementing the generator as an executable Java™ program and the process of testing the generator for functional and behavioural correctness.

The subdomain of Procedural Content Generation presents challenges that extend beyond pure software engineering, but the outcome of this project suggests that it is possible to implement general purpose content generators for non-aesthetic video game content, paving the way to greater acceptance of this technique within industry.

Acknowledgements

I offer my gratitude to Dr. Ian Bradford for supervising this project and to Dr. Olenka Marczyk for guidance with the contents of this report. I would also like to thank the academics and other teaching staff at the University of Hertfordshire that have educated and supported me throughout my degree in Computer Science.

Trademark Notices

Contents

Figures

Extracts

Tables

Appendices

1 Introduction

As noted by both Hendrikx et al. (2013) and van der Linden et al. (2014), Procedural Content Generation (PCG) is a specific application of computing where multimedia content that would normally be produced by a human designer is instead generated algorithmically by using a stochastic, pseudo-random computer program.

1.1 Background

The majority of research and development into PCG appears to have traditionally taken place in the context video games, owing to how these are arguably the most content reliant form of computer software and therefore stand to gain the most benefit from PCG. This report therefore focusses primarily on the use of PCG in the production of content for video games.

One application of PCG in serious video games, which have goals other than the entertainment of the user (Zyda, 2005), has been in physical rehabilitation, where PCG is used to adaptively change the difficulty of the game based on the player's performance and their own goals for rehabilitation (Dimovska et al., 2010).

Applications outside of multimedia are also known to exist. For example, Vanegas et al. (2011) present a procedural generator for urban parcels (groups of buildings) around roads, which can be used as part of urban city planning.

1.1.1 Formal Definition

Doull (2008) defines PCG as "the programmatic generation of game content using a random or pseudo-random process that results in an unpredictable range of possible game play spaces". The term "content" in this context refers to all the elements that compose a video game, from low-level elements such as textures and sound effects up to high-level elements such as levels and narrative plotlines (Hendrikx et al., 2013). Traditionally, this content has been created manually by multidisciplinary teams of people (Roden & Parberry, 2004; Hendrikx et al., 2013).

This definition reveals several defining aspects of PCG for video games:

1. Some or all of the content in the game is **generated automatically by the computer** with little to no intervention from human designers. It is typical to give the designers some form of parameterised control over the generation process (van der Linden at al., 2014; Shaker et al., 2015).
2. This content is generated using a **pseudo-random computer program**. This means that the algorithm used to produce the content should do so in a way that draws values at random and uses these to decide the precise form that the generated content takes. Notice how the notion of "true" randomness may also be applied here as opposed to only using deterministic pseudo-randomness.
3. The generator is capable of producing a **wide variety of content of a given type** owing to its stochastic behaviour. That is, a single content generator will typically provide unique content every time it is executed, because the sequence of pseudo-random values governing how the content is constructed is likely to be different each time it runs.

1.1.2 Types of Generation Algorithms

Togelius et al. (2010) make a distinction between generation algorithms that use a constructive approach, where the process is carefully controlled to ensure that the final result is always acceptable, and those that use a generate-and-test approach, where it is necessary to check the validity of the final result according to a set of rules and regenerate it if it isn't suitable. A dungeon environment in particular is normally considered unsuitable if it isn't possible to reach all parts of that environment without having to pass through solid walls (Doull, 2008).

Shaker et al. (2015) list four major approaches to creating dungeons, which are briefly summarised as follows:

- Space partitioning, which iteratively subdivides the available space to produce rooms that are then connected together with corridors,
- Agent-based dungeon growing, which simulates a single agent digging tunnels and creating rooms as it moves around,
- Cellular automata, which produces a dungeon by simulating life rules for each cell over a number of steps (Johnson et al., 2010), and
- Grammar-based dungeon generation, which creates dungeons by using formal graph grammars (Adams, 2002).

Finally, Buck (2011) documents eleven separate algorithms for generating mazes, which can be thought of as dungeons that do not include rooms (Procedural Content Generation Wiki, 2016). Future expansions to the system could include the implementation of one or more of these algorithms as discussed in section 5.1.4.

1.1.3 Three Dimensional Dungeon Generation

Doull (2008) elaborates on how generation of game levels in a two dimensional plane is a topic that has received a large amount of academic interest in previous years, which is most likely due to the natural susceptibility of dungeons to PCG (van der Linden at al., 2014). However, Doull also discusses an absence of generators for three dimensional game levels, caused by the difficulty of including gameplay constraints in such generators and the additional computational power needed to generate three dimensional environments.

1.2 Project Motivation

In their assessment of PCG for games, Hendrikx et al. (2013) comment on how the amount of game content expected by players is reaching unprecedented levels as technology continues to advance. They then declare that this is placing additional stress upon game designers, artists, programmers, audio engineers and other content producers to create the amount of content required for each game, and they argue that it is now reaching the point where it is becoming infeasible for all of this content to be produced manually within an acceptable period of time.

Reflecting on this manual design as a "bottleneck" for project budgets and delivery times, Hendrikx et al. then go on to propose PCG as a potential solution by mitigating some of the creational process to computers, thus decreasing the workload placed on human producers and reducing production costs to realistic levels. This proposition forms the motivation behind this project; PCG has the potential to allow the games industry to expand further, since it stands to ease the cost of producing game products of the size that is now becoming necessary.

Furthermore, Togelius et al. (2013) highlight three high-level goals for future research in PCG, followed by a number of challenges that aim to work towards these goals. The third of these challenges identifies a lack of generic "plug-and-play" content generators for video games; the majority of current generators exist to generate just one type of content for one specific game, and this may be restricting adoption of PCG due to the capital investment needed to construct a bespoke generator for each individual product. SpeedTree (2016), which produces pseudo-random vegetation, is the only general purpose generator currently in widespread use, and there is a need for generic generators of non-aesthetic content (Togelius et al., 2013).

This project represents an attempt to address this to some extent through the production of a generator that can be applied to any arbitrary game, although it still only generates a single type of game content.

1.3 Project Aims and Objectives

The project aims and objectives as they were given in the detailed proposal for this project are provided in appendix A.

The high level goal in this project was to reinforce my ongoing studies in object-oriented software engineering by specifying the generator in an extensible, object-oriented way, making use of the best practices of object-orientation, such as polymorphism and design patterns, where appropriate. The stochastic nature of the system also offers an opportunity to trial an alternative

approach to system testing, since exhaustively testing every possible output would be impractical (section 4.2).

The lower level goal was to design and implement a computer algorithm that can be used to generate dungeon environments procedurally. As the project progressed, this goal was later expanded to include the implementation of a general purpose dungeon generator that could be extended with any number of additional algorithms to generate environments for any arbitrary client system. This change was made because multiple research papers concerning PCG identify general purpose generation as an area in need of attention (Doull, 2008; Hendrikx et al., 2013; Togelius et al., 2013).

Finally, my own personal goal through this project was to gain experience of this specific application of computing, which I view as a challenging and intriguing area of study.

An assessment of the extent to which these aims and objectives were actually met is provided in the conclusion to the project in section 5.

1.4 Thesis Statement

The primary objective of this project is to investigate the feasibility of creating a procedural content generator of dungeon environments, which is referred to as the "artefact" of the project. This artefact is intended to be general purpose and usable by any client system.

This report details the practical challenges that were met when implementing this artefact, how these challenges were resolved and what knowledge could be drawn from them. It also provides an examination of how effective it is to use a multi-agent simulation to generate dungeon environments.

The remainder of this report is structured into the following major sections:

- Design, which discusses the initial design of the artefact (with supporting documents in appendices B, C and D),
- Implementation, which documents the actual practical work that took place to implement the artefact as a computer program (shown running in appendix E), and
- Testing, which documents the unit, component and system testing carried out against the artefact during and after implementation (with unit test results given in appendices F, G and H, and eight sample dungeons given in appendix I).

The report then concludes with a summary of the findings from the activities carried out during these stages, an encompassing evaluation of the project as a whole and suggestions for future improvements. Subsections are used throughout this report to separate topics for discussion.

2 Design

The artefact was designed in an object-oriented fashion, where the system is decomposed into self-contained object classes with well-defined responsibilities. The class design was done in Unified Modelling Language (UML) using ArgoUML,[1] and the full initial design of the artefact prior to implementation is shown in appendix B.

2.1 Class Architecture

A key goal when designing the artefact was to ensure that it could easily be integrated into any arbitrary client program in the form of a subsystem or middleware component, which formed the main reasoning behind using an object-oriented approach.

The artefact is divided into two major components: the coordinate package, which handles the representation of the coordinate space that dungeons are created in, and the algorithm package, which handles the specification, creation and execution of the dungeon generation algorithms themselves.

2.1.1 Coordinate Package

The artefact models a virtual environment of a regular 2D grid of cells, where each cell can either be 'filled in' or 'empty'. This representation was chosen due to the abundance of existing algorithms that also represent the environment in this way (Procedural Content Generation Wiki, 2015).

The original design introduced a "Field" class, which retains a two dimensional grid of Boolean values indicating the state of each cell. Later, these Booleans were replaced by instances of a "Cell" class that records the filled state of each cell and can be extended in the future to add new properties about cells. This led to the initial class design shown in Figure 2-A.

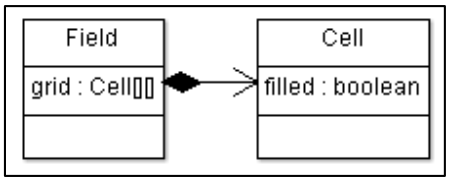

Figure 2-A: The initial coordinate space class design

A number of practical considerations were made to arrive at the final design for this part of the system, as described in the following subsections.

2.1.1.1 Client Coupling

A large amount of attention was given to the degree of coupling between the classes defining the coordinate space and their clients. Two approaches were considered to allow clients to access the field:

- Given a point in space, the Field could return a Boolean value representing the current state of the Cell at that point. It would also expose a mutator method for setting the state of a Cell at a given point. This is equivalent to data coupling; the client of Field needn't have any knowledge of the Cell class.
- Given a point in space, the Field could return a reference to the actual Cell object at that point, allowing the client to read and set the state of that Cell directly. This is equivalent to stamp coupling.

It was ultimately decided to use the latter of these two approaches. While this directly couples clients of the Field class to the Cell class and is therefore sub-optimal design,[2] it appears to be

[1] ArgoUML is an open-source modelling tool for UML 1.4. See: http://argouml.tigris.org/

[2] More formally, this design breaches the Law of Demeter, because clients of the Field class must now also interface directly with the Cell class.

the more maintainable approach, since it allows Cell to be updated with new features without the need to create corresponding methods in Field.

2.1.1.2 Two Dimensional and Three Dimensional Spaces

Integration of three dimensional coordinate spaces was considered for this project in response to the discussions by Doull (2008) noted in section 1.1.3. The eventual decision was to disregard this because it was seen as being too complex to achieve within the time available for the project, but this consideration did see the creation of a Space interface to support the addition of new coordinate spaces in the future.

It was also recognised that a Space can be thought of as a collection of Cell objects that can be iterated over, so this interface was made to extend the "Iterable" interface from the Java™ class library, allowing implementers of Space to be the target of Java's "for-each" construct.

This resulted in the class architecture shown in Figure 2-B below.

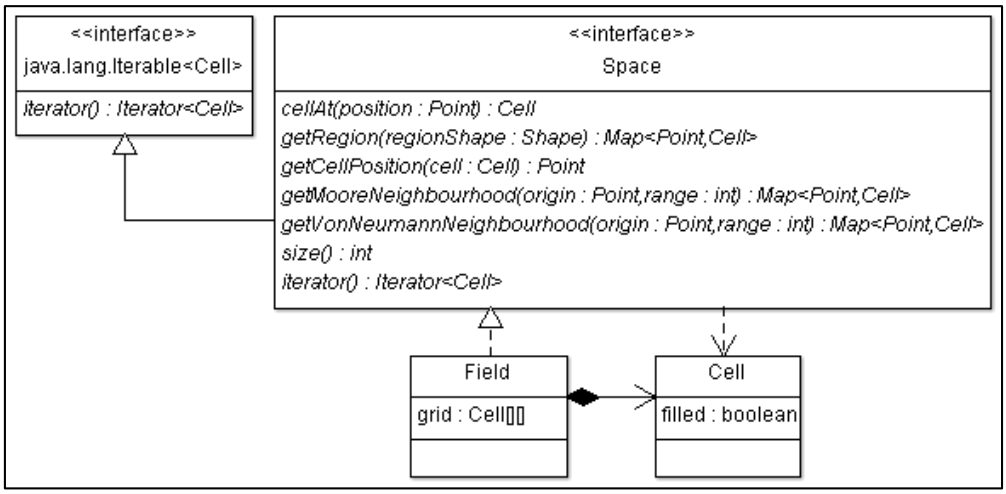

Figure 2-B: Introduction of the Space interface

The Space interface defines methods for all interactions with coordinate spaces irrespective of their dimensionality. The Field class implements these to represent a 2D coordinate space.

This architecture could potentially be extended in the future to include three dimensional environments modelled as structures of 3D cells or "voxels" (Procedural Content Generation Wiki, 2015), with each voxel being represented by an instance of the Cell class. Further discussion of this possibility is provided in section 5.2.

2.1.1.3 Coordinate Representation

The initial design used the Point class from the Java library to represent 2D coordinate locations. A later review of the design concluded that this wouldn't be acceptable for this application because instances of Point are mutable, making them unsuitable for use as map keys as shown in some of the methods in the Space interface in Figure 2-B.

As a solution, a custom Coordinate2D class was designed instead, which represents an immutable coordinate location in space. Implementing Coordinate2D as a custom class also provides more control over how coordinates are stored and handled. This led to the architecture in Figure 2-C below.

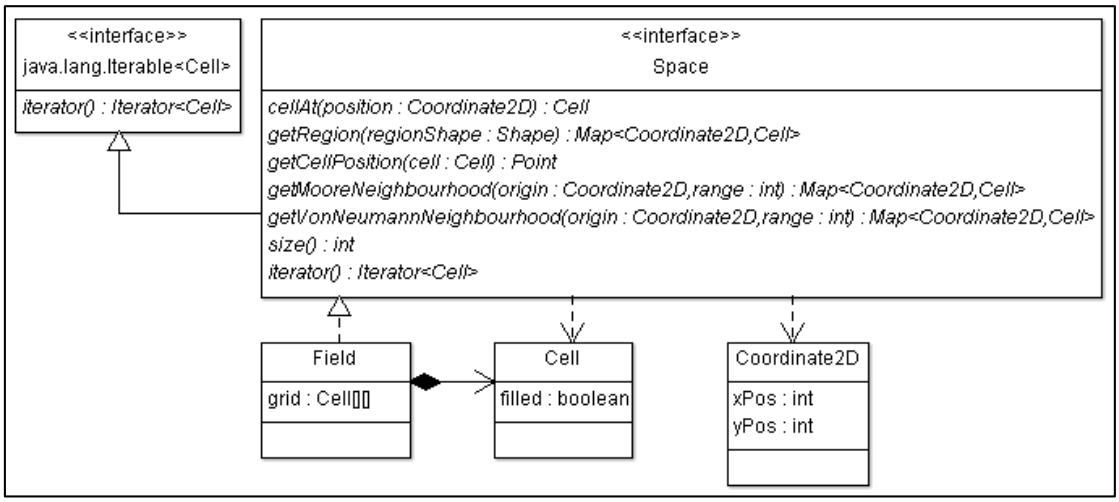

Figure 2-C: Introduction of the Coordinate2D class

2.1.2 Algorithm Package

To support the addition of future algorithms, the decision was made to model each algorithm as a class in its own right, which would interact with the coordinate space and manipulate it in a fashion specific to that algorithm. This would make it possible to easily add new algorithms in the future by creating a new class programmed to the same interface.

For this reason, the strategy design pattern was employed: an interface named "Algorithm" was defined that would contain a single abstract method to act upon a given space in a way defined by the subtype. Each specific algorithm would then implement this interface, allowing new algorithms to be added as realisers of this strategy.

This design is shown on the left in Figure 2-D below, which contains two algorithms discussed later in this report. The associated academic definition of the strategy design pattern is shown on the right for comparison.

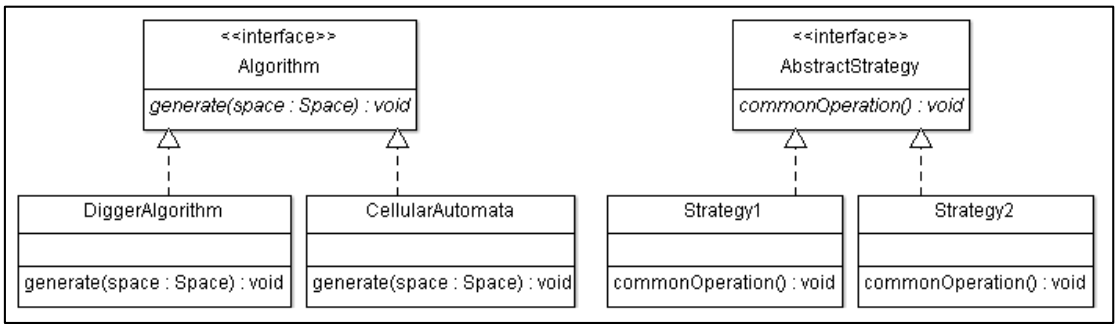

Figure 2-D: The strategy pattern for the generation algorithms

To remove the need for clients to instantiate algorithms directly, this design was then expanded into the factory design pattern. Here, each subtype of Algorithm has a corresponding "AlgorithmFactory" object responsible for instantiating instances of that algorithm. This is shown in Figure 2-E below.

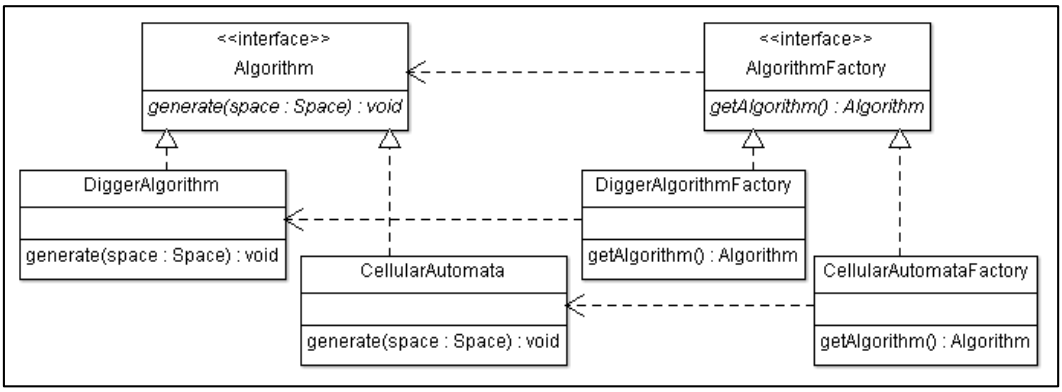

Figure 2-E: The factory pattern for algorithm instantiation

While this class design formed the basis of the design eventually used in the final artefact, it wasn't possible to use the factory pattern in practice, as discussed in section 3.2.2.1.

2.1.3 Middleware Interfaces

As stated in section 1.3, a goal of the project was to produce a content generator that can be used by any arbitrary client computer program. This means that client systems should be able to use this artefact to generate content for them on their behalf.

2.1.3.1 Programmatic Interface

The most important interface is arguably the one between the artefact and the client system using it. The initial design for this interface involved generating a dungeon and then converting it into a sequence of byte values, with each byte representing a single cell. This sequence could then either be passed directly to the client system or written to a file to be used later.

After further consideration, this design was refined to take advantage of the API facilities of Java. Instead of outputting a sequence of bytes, the Java classes making up the artefact would be exposed via a public API, allowing client systems to instantiate the classes and call methods on them directly. This benefits the client system by providing more control over the generation process and producing more maintainable code.

To simplify client access to the artefact, a top level "Generator" facade class was designed with the goal of providing a single, well-defined point of access to the system. The intention was that the client would only have to interact with this class in order to utilise the artefact, although in practice, it is necessary to interact with other classes as well (section 3.2.3.1).

2.1.3.2 Graphical User Interface

A Graphical User Interface (GUI) to the generator was also designed to allow a user to manually configure and preview the generated dungeons. Normally, such an interface wouldn't be provided, since the generator would only need to be used programmatically as an API in order to integrate it with a client system. However, a GUI has been provided in this project for the sake of demonstration. It could also be argued that the graphical user interface makes it possible for programmers to fine-tune the configuration parameters for their chosen algorithm before using them in their own systems.

The GUI allows the user to select and configure an algorithm and observe the environments created by it. A digital scan of the final hand-drawn design is shown in Figure 2-F; other designs predating this one were laid out differently but featured the same components.

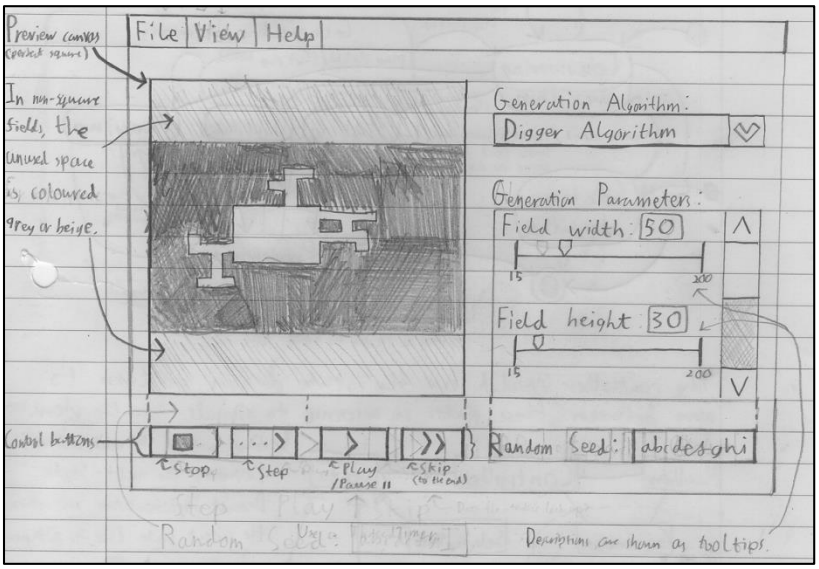

Figure 2-F: Final graphical user interface design prior to implementation

The behaviour of this GUI was designed using the state design pattern, where transitions between states would enable and disable components as illustrated in Figure 2-G below.

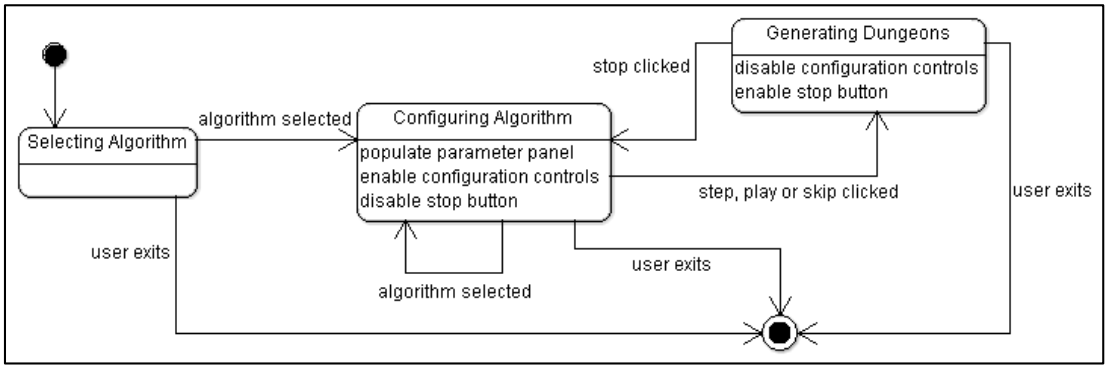

Figure 2-G: Basic state diagram for the graphical user interface

2.2 Digger Algorithm Design

A secondary target of this project was the design and implementation of a dungeon generation algorithm of my own design, known simply as the "digger algorithm". This algorithm attempts to simulate a large number of independently acting agents termed as "diggers" that move around the environment according to a basic set of rules and carve out rooms and corridors as they go.

It must be noted immediately that this section describes the original design for the algorithm. Modifications to the precise details of the algorithm became necessary while it was being implemented in order to compensate for practical constraints, which are highlighted and discussed in detail in section 3.3.

The parameters and description for this algorithm are listed in appendices C and D respectively.

2.2.1 Inspiration

There were two major sources of inspiration for the design of this algorithm, as presented below.

2.2.1.1 Anderson's Dungeon Building Algorithm

This algorithm was largely inspired by a similar algorithm developed by Anderson (2014) for a specific video game, which builds a single dungeon from an initial staring room by iteratively adding "features" to it (rooms, corridors, traps, etc.) until a particular set of conditions has been reached, such as the final size of the dungeon.

2.2.1.2 Agent-Based Models

The design of the digger algorithm was also heavily influenced by the practice of agent-based modelling from artificial intelligence. The essential premise of agent-based modelling is that a large number of agents following a simple set of rules influence the state of the environment around them, which in turn influences how those agents behave. Complex patterns and behaviours may then emerge from this bidirectional interaction between the agents and their environment.

Shaker et al. (2015) document an agent-based generation algorithm that only uses a single agent but otherwise behaves in a largely similar manner, as discussed in section 1.1.2, although this algorithm did not form the initial inspiration for using an agent-based approach.

2.2.2 Configuration Parameters

As stressed by van der Linden et al. (2014), it is important that designers have some means of configuring the behaviour of procedural content generation algorithms to "steer" the final results towards a desired outcome. Therefore, the digger algorithm was designed to be configurable by the client using a range of parameter values (provided in appendix C) that inform the rest of the generation process.

2.2.3 Algorithmic Steps

The full algorithm was expressed in natural language pseudocode before any practical implementation started, as shown in appendix D. The goal at this stage of the development process was to establish how the algorithm would function without concern for its implementation.

A key element of the algorithm design is the selection of rules controlling where diggers are permitted to go based on the state of their Moore neighbourhood. Consider Figure 2-H below. The agent depicted here may only move into the cell ahead of it (in this case, the northern cell) if all of the following cases apply:

1. The northern cell is within bounds and is filled in,
2. Either or both of the western and north-western cells are out of bounds or filled in, and
3. Either or both of the eastern and north-eastern cells are out of bounds or filled in.

The checked cells are shown shaded in blue below and are rotated around the digger for each orthogonal direction it can move in.

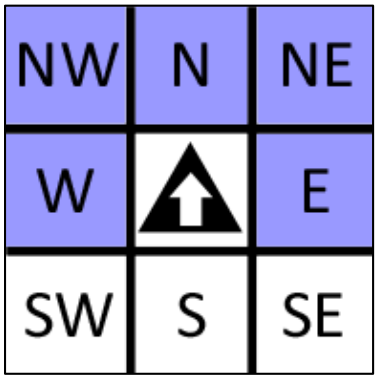

Figure 2-H: Depiction of a northward facing digger agent within its Moore neighbourhood

These constraints aim to ensure that the agents construct recognisable dungeons as they move by avoiding the formation of corridors that are more than one cell in width, leveraging the concept of the agents being influenced by their environment. Furthermore, diggers are only permitted to construct rooms in rectangular regions where there are no empty cells.

2.2.4 Algorithm Class Design

This algorithm was modelled as a self-contained class for the reasons described in section 2.1.2. Additionally, it was also a natural fit to model the digger itself as a "Digger" class, since each

digger possesses its own independent state at a given point in time and has behaviour associated with it. A Digger maintains its position, orientation, age and predetermined lifetime as per the description in appendix D. The concept of an agent was also separated into an "Agent" interface implemented by the Digger class, to ease the introduction of new types of agent in the future.

To represent the digger's orientation, an enumeration called "Direction" was created that contains the four cardinal directions (NORTH, EAST, SOUTH and WEST). While this restricts the digger to a two dimensional plane, this is viewed as acceptable since this algorithm is only designed to function in two dimensions.

These design considerations ultimately led to the architecture shown in Figure 2-I below. The DiggerAlgorithm class corresponds to the class of the same name in Figure 2-E.

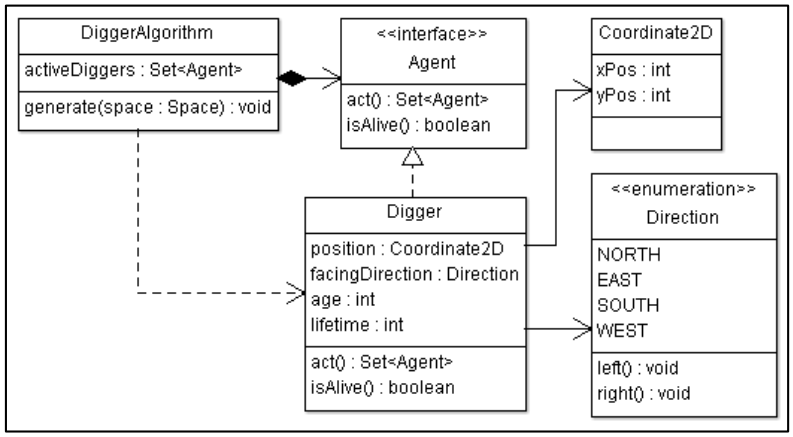

Figure 2-I: The class architecture for the digger algorithm

3 Implementation

The artefact was implemented in the Java™ programming language using NetBeans.[3] The full codebase for the artefact is provided alongside this report, including documentation describing the individual classes.

3.1 Language Choice

Using Java is beneficial because a goal for the project is to make the artefact as maintainable and portable as possible. Programs written in Java are packaged into platform independent byte-code that can be executed on any operating system with an appropriate Java Runtime Environment. The procedural dungeon builder algorithm by Anderson (2014) introduced in section 2.2.1.1 was originally implemented in the Java language, providing confidence that Java is suitable for this domain in practice.

An original consideration was to use the C# programming language, which is also object-oriented and includes many features that make it similar to Java (Radeck, 2003). However, C# is viewed as being less portable than Java because it is compiled into platform specific machine code that can then only run on one platform (Albahari, 2000), making it less suitable for this project.

3.2 Model Implementation

This section describes the activities that were carried out to implement the main core of the generic dungeon generator, whose design is documented throughout section 2.1.

3.2.1 Coordinate Space Implementation

The original design for the classes representing the coordinate space as discussed in section 2.1.1 was mainly upheld during the implementation phase. The details of the changes that were made are discussed below.

3.2.1.1 Neighbourhood Detection

The original specification for the Space interface as it appears in Figure 2-C features a pair of methods that allow a client class to acquire a certain neighbourhood of cells around a given cell. The first of these methods would find and return the Moore neighbourhood around a given point while the second would find and return the von Neumann neighbourhood around a given point. These methods are shown in Extract 3-A below.

```
public abstract Map<Coordinate2D, Cell>
  getMooreNeighbourhood(Coordinate2D origin, int range);
public abstract Map<Coordinate2D, Cell>
  getVonNeumannNeighbourhood(Coordinate2D origin, int range);
```

Extract 3-A: The signatures of the original methods for acquiring two types of neighbourhood from a Space

The computational steps involved in each of these methods were found to be very similar, so they were therefore combined into one. The condition defining each neighbourhood was delegated to a "NeighbourhoodType" enumeration as illustrated in Extract 3-B and Extract 3-C.

```
public Map<Coordinate2D, Cell> getNeighbourhood(Coordinate2D
  origin, int range, NeighbourhoodType type);
```

Extract 3-B: The signature of the refactored neighbourhood method

[3] NetBeans is an integrated development environment for multiple programming languages, including Java. See: https://netbeans.org/

```
public enum NeighbourhoodType {
  VON_NEUMANN {
    @Override
    public boolean isValid(int distance, int radius) {
      return (distance > 0 && distance <= radius);
    }
  },
  MOORE {
    @Override
    public boolean isValid(int distance, int radius) {
      return (distance > 0);
    }
  };
  public abstract boolean isValid(int distance, int radius);
}
```

Extract 3-C: The implementation of the NeighbourhoodType enumeration

This approach means that the rules governing the members of a neighbourhood are encapsulated in their own class, granting the ability to add new types of neighbourhood and modify existing ones without having to change the method in Extract 3-B.

3.2.1.2 Interconnectedness

There is no guarantee that a dungeon created with a particular algorithm will be fully interconnected (as indicated in section 1.1.2), so an additional method was added to the Space interface and defined in the Field class to allow clients to determine whether the dungeon is fully interconnected. The implementation of this method performs a breadth-first traversal of the dungeon to check whether all of the empty cells in the space are connected to one another (either directly or indirectly) and returns true or false accordingly.

This method actually originates from the testing framework covered in section 4.2 below, but because the method acts on the contents of the space, it was considered better design to define it within the Space interface itself, since this would decrease coupling and allow all other clients of Space to use this method as well.

3.2.1.3 Value Overflows and Underflows

When designing the coordinate system, it was assumed that each coordinate position would be expressible as any integer value ranging between $-\infty$ and ∞. However, because Java uses signed 32-bit integers, its integer datatype is limited to holding values ranging between -2^{31} and $2^{31}-1$ inclusive.[4]

This limitation meant that a new potential fault had to be considered when implementing the Coordinate2D class, because if a coordinate position is translated above the maximum value or below the minimum value, it wraps around to the other extreme.

To overcome this, a defensive check was included in the "translated()" method of Coordinate2D to throw a custom exception if the translation would cause an overflow or an underflow, as illustrated in Extract 3-D below.

[4] These limits are stored in the MIN_VALUE and MAX_VALUE constants of the Integer class respectively.

```
public Coordinate2D translated(int deltaX, int deltaY) {
  if (
     (deltaX > Integer.MAX_VALUE - xPos ||
      deltaX < Integer.MIN_VALUE + xPos)||
     (deltaY > Integer.MAX_VALUE - yPos ||
      deltaY < Integer.MIN_VALUE + yPos)
  ) {
     throw new ValueOverflowException(/* exception message */);
  }
  return new Coordinate2D(xPos + deltaX, yPos + deltaY);
}
```

Extract 3-D: Synopsis of the "translated()" method of Coordinate2D

The logic shown here was actually found to be incorrect during testing, as discussed later in section 4.1.3.

3.2.2 Algorithm Package Implementation

The algorithm package was implemented according to the design in section 2.1.2, with the following changes.

3.2.2.1 Algorithm Instantiation

The goal of the original design was to offer support for an arbitrary number of algorithms represented as classes and to use the factory design pattern to separate the instantiation of these algorithms from their definitions. However, a key element that mistakenly wasn't considered during the original design was the fact that each algorithm would need to be able to accept parameter values, to allow users of the algorithms to direct their behaviour as indicated in the formal definition in section 1.1.1.

In practical terms, this meant that the algorithms would need to be able to accept a range of parameters in their constructors. Unfortunately, this made it impossible to use the factory design pattern, because each factory relies on its product having a constructor that accepts no parameter values. This part of the system therefore had to be refactored during development to accommodate this.

3.2.2.1.1 Reflective Factory

The first approach to solving this problem was to use Java's Reflection facility to create a generic algorithm factory, which would dynamically reflect on a given algorithm to discover its range of parameter values and allow the user to configure these at run time. To support this, a Java annotation type called "ParamInfo" was created that would be applied to each parameter value in the algorithm's constructor to define its natural name, description and maximum, minimum and default values, as illustrated in Extract 3-E below.

```
DiggerAlgorithm(
  @ParamInfo(
     fullName="Field width",
     description="The width of the field in which dungeons will
be built.",
     lowerLimit=10,
     defaultValue=20,
     upperLimit=200) int fieldWidth,
     /* further parameters */
) { /* object initialisation */ }
```

Extract 3-E: Marking algorithm parameters with the ParamInfo annotation

While this approach did function in preliminary tests, it was ultimately discarded because Java Reflection was seen as unsuitable for tasks outside of testing and interacting with foreign Java classes whose interfaces are not known in advance.

3.2.2.1.2 Parameter Encapsulation

The next approach was to attempt to encapsulate the parameters into their own "NamedParameter" class, an instance of which would store the metadata for a particular parameter as well as its current value. This class originally used generics to define the type of numeric value it held, giving it the signature shown in Extract 3-F.[5]

```
public final class NamedParameter<T extends Number & Comparable>
{
  private final String name, description;
  private final T lowerLimit, upperLimit;
  private T value;
  public T getValue() { /* ... */ }
  public void setValue(T newValue) { /* ... */ }
  /* constructor and additional accessor and mutator methods */
}
```

Extract 3-F: Synopsis of the original NamedParameter class

Each algorithm factory would then keep a collection of NamedParameter objects representing the parameters needed by its product algorithm. It would also implement the two methods shown in Extract 3-G below to allow clients to acquire the parameters and create the product algorithm.

```
public abstract NamedParameter[] getParameters();
public abstract Algorithm createAlgorithm(NamedParameter[]
  params);
```

Extract 3-G: Proposed signatures for the abstract algorithm factory

The disadvantage of this approach is that the client class would have to acquire the NamedParameter objects using the accessor method, set their values directly and then pass them back via the factory method, which is a largely unintuitive process. Furthermore, because the "getParameters()" method in Extract 3-G returns un-typed NamedParameter objects, the client would have to manually downcast the Number returned by the "getValue()" method of each one to a specific numeric type, thus creating coupling by requiring preliminary knowledge of the datatypes used. Finally, under this approach, clients would also have to access and manipulate NamedParameter objects directly via the builder, which is undesirable program design.[6]

3.2.2.1.3 The Builder Pattern

The final approach was to replace the factory design pattern with the builder design pattern, where products are created incrementally rather than in one single operation (Metsker and Wake, 2006).

Each factory was replaced by a builder that records the parameters used by the product algorithm, exposes accessor and mutator methods for them (as illustrated in Extract 3-H below) and constructs and returns a new instance of this algorithm using these parameters when asked to do so by the client.

---

[5] In this extract, the generic type T represents a subclass of Number that also implements the Comparable interface. Only immutable Numbers implement Comparable in order to avoid errors caused by concurrent updates to the value during comparison, as described here: http://stackoverflow.com/q/480632. Notation source: http://stackoverflow.com/a/9788113.

[6] This breaches the Law of Demeter because clients of the builder must interface with NamedParameter.

```
public int getFieldWidth() { /* ... */ }
public void setFieldWidth(int fieldWidth) { /* ... */ }
public int getFieldHeight() { /* ... */ }
public void setFieldHeight(int fieldHeight) { /* ... */ }
/* ...and so on for the remaining parameters. */
```

Extract 3-H: Examples of the explicit accessor and mutator methods exposed by the DiggerAlgorithmBuilder class

The NamedParameter class was also simplified by removing its generic type and storing all parameters as integer values, on the basis that the majority of the parameters used in the implemented algorithms were integers.[7]

This resulted in the final class architecture for algorithm instantiation shown in Figure 3-A below. Contrast this to Figure 2-E in section 2.1.2, which shows the original factory design.

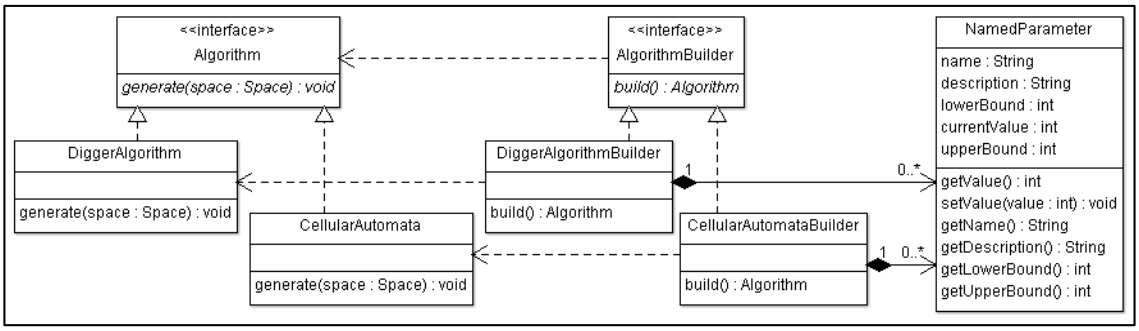

Figure 3-A: The builder pattern for algorithm instantiation

This appeared to be a better fit for the problem at hand, because the client would now have the option of only setting those parameter values they are interested in and leaving the others at their default values.

The remaining problem with this design was the fact that the accessor and mutator methods defined by the non-abstract builders were unique to those builders. This meant that clients would have to refer to specific subclasses of AlgorithmBuilder to acquire and set parameter values, thus increasing coupling and reducing extensibility.

The solution to this was to convert the AlgorithmBuilder interface into an abstract class that maps the name of each NamedParameter object to that NamedParameter. It exposes the implemented methods shown in Extract 3-I to allow clients to access and modify the values of the parameters based on their names, replacing the previous approach shown in Extract 3-H. The subtypes of AlgorithmBuilder would then populate the parameter map in their constructors.

```
public final NamedParameter getParameter(String name) {/*...*/}
public final void setParameter(String name, int value) {/*...*/}
```

Extract 3-I: The generic accessor and mutator methods exposed by the AlgorithmBuilder class

An advantage of this approach is that clients no longer have to address subtypes of AlgorithmBuilder directly, because all of the required methods are defined in the super class. Meanwhile, a clear disadvantage is that strings aren't type-safe. Further discussion of this matter is provided in the conclusion in section 5.1.2.

3.2.2.2 Algorithm Template Generation Methods

The prior definition of the abstract Algorithm class in section 2.1.2 contained a single method for running the generation process, shown in Extract 3-J below. Subclasses would override this

[7] To support normalised percentages ranging between 0.0 and 1.0, a "NamedPercentageParameter" class was also created as a subclass of NamedParameter. This subclass can accept a normalised percentage value in its constructor and then converts this into an un-normalised integer ranging between 0 and 100.

method to contain the actual logical steps of the algorithm and act upon the coordinate space given as a parameter.

```
public abstract void generate(Space space);
```

Extract 3-J: The original abstract generation method declared in the abstract Algorithm class

However, this design is insufficient because the user needs to be able to progress through the algorithm one step at a time, whereas the method shown above would only allow the algorithm to be run to completion as a single atomic task.

For this reason, the method above was broken down into the three separate methods shown in Extract 3-K below. Additionally, the coordinate space is now accepted in the constructor of the Algorithm class and stored in a field of that class.

```
protected abstract void setUp();
protected abstract void mainLoopStep();
protected abstract boolean isFinished();
```

Extract 3-K: The final three abstract generation methods declared in the abstract Algorithm class

These methods may be summarised as follows:

- The "setUp()" method is invoked once by the client at the start of the algorithm.
- The "mainLoopStep()" method is invoked by the client to progress the algorithm by one time step.
- The "isFinished()" method is implemented to indicate whether or not the algorithm has reached its terminal state.

The abstract Algorithm class defines two public methods and one private method that use these abstract methods, as illustrated in Extract 3-L below (calls to the abstract methods are shown in bold). This uses the template method design pattern; the Algorithm class defines the common logic for initialising, repeating and terminating the algorithm while its subclasses are responsible for defining the actual steps of the algorithm.

```
private void reset() {
  for (Cell c : getSpace()) {
    c.setFilled(true);
  }
  setUp();
}
public final synchronized void generateStep() {
  if (isFinished()) {
    reset();
  } else {
    mainLoopStep();
  }
}
public final synchronized void generate() {
  do {
    generateStep();
  } while (!isFinished());
}
```

Extract 3-L: The generation methods defined in the abstract Algorithm class

3.2.3 Interface Implementation

The implementation of the two interfaces to the system proceeded largely according to the original designs given in section 2.1.3, with minor differences as discussed below.

3.2.3.1 Programmatic Interface

To support access to the generator by client systems, a top level "Generator" facade class was planned to be implemented as described in section 2.1.3.1, which the client would be able to interface with to interact with the underlying subsystem.

This, however, also underwent significant changes during development. Initially, this class was implemented as a singleton[8] that mapped enumerated literals representing each algorithm to an instance of the builder used to create instances of that algorithm.

Later on, this class was simplified by implementing it as an enumeration known as "GeneratorHeader" in the final artefact, where each literal represents a single algorithm and stores both the natural name of the algorithm and an instance of the builder that can be used to create instances of that algorithm. The architecture of the GeneratorHeader class is shown in Figure 3-B, and the implementation of its literals is shown in Extract 3-M. The Algorithm and AlgorithmBuilder classes in Figure 3-B correspond to those of the same names in Figure 3-A.

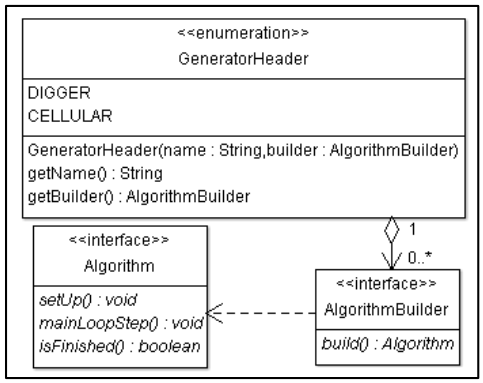

Figure 3-B: Implementation of the GeneratorHeader facade

```
public enum GeneratorHeader {
  DIGGER("Digger algorithm", new DiggerAlgorithmBuilder()),
  CELLULAR("Cellular automata", new CellularAutomataBuilder());
  // Further algorithms can be added in the same way.
  /* constructor, fields and methods */
}
```

Extract 3-M: The implementation of the literals in GeneratorHeader

Clients can then interact with the underlying subsystem as illustrated in Extract 3-N below.

```
AlgorithmBuilder builder = GeneratorHeader.DIGGER.getBuilder();
builder.setParameter("Field width", 50);
builder.setParameter("Corridor branch probability", 20);
/* further calls to setParameter() as required */
Algorithm alg = builder.build();
/* 'alg' can now be used to generate dungeons */
```

Extract 3-N: An illustration of how a Java client might access the system programmatically

This increases coupling because the creation of a new algorithm and associated algorithm builder would require the addition of another literal to this enumerated class. However, this task isn't viewed as being particularly time consuming or error prone.

[8] In the singleton design pattern, only one instance of the class ever exists during the execution of the program. All clients access and share the same instance rather than instantiating their own instances.

3.2.3.2 Graphical User Interface

The graphical user interface described in section 2.1.3.2 was implemented using the Model-View-Controller design pattern. This means the GUI was decomposed into two class packages, one containing view related classes and another containing controller classes. The underlying subsystem that actually generates the environments is treated as the model, which is accessed via the interface described in section 3.2.3.1 above.

The class architecture of the GUI is shown in Figure 3-C, and two digital screenshots of it are given in appendix E. The implementation details are not covered here as they do not contribute towards the core aims of the project, but note the following:

- The controller classes were implemented as per the state design pattern, where transitions between states would enable and disable certain components in the view according to the state diagram in Figure 2-G.
- Graphics drawing is achieved using a technique similar to the double buffering design pattern (Nystrom, 2014) to ensure that refreshing the preview of the dungeon is carried out as a single atomic operation.
- The ParameterPanel class uses the observer design pattern. NamedParameter is an observable that notifies its ParameterPanel when its value changes, ensuring that the graphical rendering of the parameter is kept up-to-date.
- Class coupling was less of a concern for this part of the system, because the GUI is unlikely to change in the future and only serves to illustrate the underlying system.
- The GUI makes improper use of the Event Dispatch Thread, as it stops responding when algorithms are executed on larger coordinate spaces. This is also less of a concern since the GUI only serves as a tool for illustrating the system, as discussed in section 2.1.3.2.

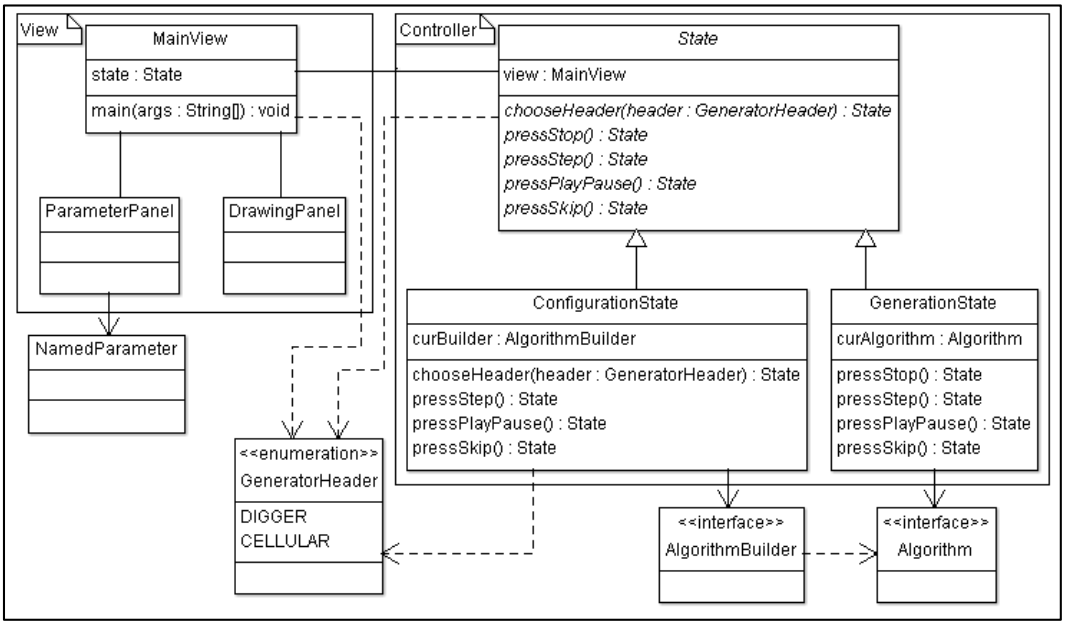

Figure 3-C: The class architecture for the graphical user interface

3.3 Digger Algorithm Implementation

As noted in section 2.2, it was necessary to make some practical modifications to the details of the digger algorithm while it was being implemented. These modifications are discussed below.

3.3.1 Origin Room Separation Distance

Originally, the first step of this algorithm as detailed in appendix D involved ensuring that all of the origin rooms created at the start of the algorithm were some minimum distance apart according to the associated parameter shown in appendix C. This aimed to avoid situations

where origin rooms could appear very close to one-another, potentially resulting in the creation of unacceptably small dungeons.

Once implementation started, it was discovered that the process of reliably satisfying this constraint would have been a difficult task to achieve. In dungeons with a relatively small field and many origin rooms, one can imagine a scenario where the available space is completely used up before all of the rooms have been placed, which would cause the algorithm to endlessly scan for an available space unless this condition was detected and prevented.

Because of the difficulty of this aspect of the algorithm, it was left to be implemented later. However, later tests with the algorithm (section 4.2) showed that it still functioned appropriately in the majority of cases even without this feature, so room separation was ultimately omitted entirely.

3.3.2 Room Construction Process

In the original algorithm in appendix D, new rooms are constructed in steps 1, 2 and 3.e. A number of changes were made to the process of actually constructing these rooms as listed below:

- In the original design, the probability that a digger attempts to construct a room immediately ahead of itself would rise linearly as it aged, eventually reaching 100% in the turn in which it expires. In the final implementation, however, this probability is instead a constant value defined by the client.
- Previously, when a new room was created, the digger agents stemming from that room would be placed in the margin around the outside of that room, embedded in the perimeter walls. This was later changed to the simpler approach of creating diggers within the perimeter of the room, facing the exterior walls.
- The original design aimed to allow the creation of irregularly shaped, non-rectangular rooms that would fit around existing empty cells in the rectangular region targeted by the digger. The complexity of this feature meant that it couldn't be implemented within the available time, but tests against the algorithm (section 4.2) suggest that it still produces suitable results without this feature.

3.3.3 Digger Agent Implementation

The digger agent is represented by a Digger class that is used by the DiggerAlgorithm class as discussed in section 2.2.4. However, as these two classes were implemented, the decision was eventually made to make the Digger class an internal private class of DiggerAlgorithm, meaning that instances of Digger would be able to directly access the private members of DiggerAlgorithm and wouldn't be usable anywhere else. While this is design isn't ideal,[9] it may be justified for the following reasons:

- The Digger class exists solely for use by the DiggerAlgorithm class and is unlikely to be reusable for any other purpose.
- The specification of how diggers behave forms part of the full algorithm.
- Diggers need access to the parameter values defined in DiggerAlgorithm to determine their behaviour.

3.3.4 Introduction of RandomSet

With every iteration of the algorithm, the living digger agents are made to act in a random order as indicated in the fourth step of the algorithm in appendix D. To support this, a "RandomSet" class was implemented as an extension to the AbstractSet class from the Java class library.

The RandomSet class allows clients to draw elements from the set at random and iterate over all of the elements in a random order, according to a given pseudo-random number generator (an instance of the Random class). Internally, RandomSet stores its elements in a list and controls

[9] Under this implementation, the Digger class is said to be "content coupled" to the DiggerAlgorithm class. This also harms cohesion because the Digger class is now reliant on the internal representation of DiggerAlgorithm.

the addition of elements to avoid duplicate entries, meaning that RandomSet may be thought of as an object adapter (Freeman et al., 2004), because it exposes one interface (a set) while actually implementing it using another interface (a list).

The RandomSet class is used in the digger algorithm for storing collections of digger agents and iterating over them in a pseudo-random order in each time step, but it has been designed to be reusable in other domains as well.

3.4 Cellular Automata Implementation

A second algorithm was also implemented to illustrate the process of extending the system by adding further generation algorithms to it. The algorithm chosen is an alteration of cellular automata, where the state of each cell depends on the state of the cells surrounding it during the previous time step (Wolfram, 2002; Shiffman, 2012; Shaker, 2015).

The cellular automata algorithm implemented into this artefact is based on the definition given by Johnson et al. (2010), and the resulting class architecture is shown in Figure 3-A. The steps involved in implementing this algorithm were as follows:

1. Create a new subclass of the abstract Algorithm class to represent the new algorithm (CellularAutomata) and then implement the template methods “setUp()”, “mainLoopStep()” and “isFinished()”, which are described in section 3.2.2.2.
2. Create a new subclass of the abstract AlgorithmBuilder class to represent the builder responsible for creating instances of this new algorithm (CellularAutomataBuilder).
3. Add a new literal to the GeneratorHeader enumeration (section 3.2.3.1) so clients interfacing with this enumeration can acquire a builder for the new algorithm. This is shown by the “CELLULAR” literal in Extract 3-M above, which is offered to clients alongside the “DIGGER” literal representing the digger algorithm.

Note that these three steps can now simply be repeated for each new algorithm added to the system, allowing the GeneratorHeader enumeration to be built up with a library of algorithms for the client system to choose from.

4 Testing

The testing was divided into unit, component and system testing as described below.

4.1 Unit and Component Testing

During development, particular attention was paid to testing the classes representing the coordinate space (sections 2.1.1 and 3.2.1), because these classes are relied upon by the generation algorithms and therefore must be known to function correctly before the algorithms can be tested.

4.1.1 Unit and Component Testing Strategy

To make the testing process as efficient as possible, the incremental testing strategy (Lethbridge and Laganière, 2005) was used to test the units and components making up the complete system. Under this strategy, classes are tested in a bottom-up fashion, meaning that if a tested class uses any number of other classes, those classes must be fully tested first. In this way, if a fault is discovered when testing a class, it is guaranteed to originate from within that specific class, since all of the classes used by the tested class are already known to work correctly. Methods within each class were also tested in this way; if a method uses other methods defined in the same class, those methods must be tested first.

The tested classes and methods were chosen based on their complexity and importance. Methods with trivial implementations (such as accessor methods that simply return a value without carrying out any processing) were viewed as being relatively unlikely to contain errors, so methods of this nature were tested indirectly by testing the more complex methods that use them. Likewise, basic classes with little complexity were tested indirectly by testing their more complex clients.

Equivalence classes (Lethbridge and Laganière, 2005) were also defined for each individual method that accepts inputs, in an attempt to further improve testing efficiency. Lethbridge and Laganière indicate that defects are more likely to occur at the boundaries between equivalence classes, so emphasis was placed on the boundaries of the identified equivalence classes when selecting test data.

4.1.2 Unit and Component Testing Process

One test plan was created for each tested class, which contains one test script for every tested method. Each test script defines one or more test instances that the method is executed on,[10] a list of equivalence classes for that method as described in section 4.1.1 above[11] and a table recording the tests performed on that method. The table includes columns for the following information:

- The list of inputs passed as actual parameter values to the method being tested,
- An indication of the expected data or behaviour,
- The actual data or behaviour that resulted from the test, and
- A record of the changes made to the implementation of the method, if any, in response to how the expected and actual outputs compare.

Each test plan was then translated into a JUnit[12] class containing one test method for each row of the table described above.

4.1.3 Unit and Component Testing Results

Three classes from the coordinate package were selected for testing. The outcomes of these tests are briefly summarised below.

[10] No test instances are defined when testing constructors.
[11] No equivalence classes are defined when testing methods and constructors that don't accept any input parameters.
[12] JUnit is a framework for performing unit testing in Java. See: http://junit.org/

1. The Coordinate2D class was tested because it has no dependent classes but a large number of client classes. The test results are shown in appendix F. Testing this class revealed a defect in the overflow detection logic introduced in section 3.2.1.3, where an exception would always be thrown when translating a coordinate point whose X and/or Y components were negative.
2. The Field class was tested because it utilises many of the other classes in the coordinate package, which are all considered to be trivial enough to not require their own explicit testing (except for Coordinate2D, which has already been tested). The results are recorded in appendix G. Testing this class led to various technical corrections to the operation of some of its methods.
3. The RandomSet class introduced in section 3.3.4 was tested in order to ensure that it obeys the standard behaviour of a set when adding and removing elements. This test also ensures that random sets with the same seed draw elements in the same (pseudorandom) order. The results are shown in appendix H; one correction was made to handle retrieving elements from empty sets.

4.2 System Testing

In traditional software engineering, a system test would involve formally verifying that the implemented system behaves in the expected way. In the context of this artefact, this would mean testing each implemented dungeon generation algorithm to ensure that the majority of the environments produced are fully interconnected.

However, because this artefact behaves in a pseudo-random way (as per the definition provided in section 1.1.1), each algorithm can produce a very large number of outputs for each selection of inputs, effectively making it infeasible to exhaustively test every single output that can be produced.

To deal with this issue, the devised scheme for performing system testing followed a less formal experimental approach, whereby instead of attempting to test every single output, a representative subset of these outputs would be generated and tested instead. The aim of this was to allow each algorithm to be tested to the point where it appears to function correctly beyond all reasonable doubt.

Note that this approach is only intended for performing quantitative tests over a large number of dungeons to verify their quantitative properties as described above. To understand the qualitative nature of the algorithms, it is still necessary to manually generate and analyse the dungeons. This is done for the digger algorithm in section 5.1.3.

4.2.1 System Testing Framework

It was anticipated that manually testing each algorithm would be a highly time consuming process, even with the reduced subset of outputs discussed above. With this in mind, an external testing framework was written to automate the majority of the process. This framework generates the required subset of outputs and then heuristically judges each output automatically, thus helping to accelerate the testing process.

An aim for this framework was to make it generic, allowing it to be applied to any algorithm regardless of how that algorithm operates. The use of object-orientation is therefore highly beneficial here because new algorithms can be added to the generation model without having to modify the testing algorithm that runs as a client of that model.

The framework calculates two properties of each dungeon:

- Its utilisation of the available space (the proportion of cells in the final dungeon that are marked as empty), and
- Whether or not the dungeon is fully interconnected (whether it's possible to navigate to all the empty cells within the dungeon as discussed in section 1.1.2).[13]

[13] This is determined by performing a breadth-first traversal of the entire dungeon; see section 3.2.1.2.

When running the framework, the user must minimally specify the name of the algorithm to test (identified by the name of the associated literal in GeneratorHeader as shown in Extract 3-M) and the number of dungeons that the framework should generate and evaluate. Testing a higher number of dungeons produces more representative results but also slows the testing process. The user may also optionally specify the output format (plaintext or Comma Separated Values), the output file (the standard output stream by default) and a configuration file as described in section 4.2.1.1 below.

4.2.1.1 Parameter Grid Searching

By default, the testing framework will only run tests using the default values of the tested algorithm. This may not be acceptable in all cases, because the tester may wish to see how certain combinations of parameter values affect the validity of the dungeons produced.

To support testing ranges of parameter values, the framework also allows its user to provide a configuration file defining these ranges. This is based on the notion of grid searching from artificial intelligence, which aims to identify the optimal set of inputs for achieving a desired output (for example, Hsu et al., 2010).

As an example, consider Extract 4-A below, which shows a configuration file for testing the digger algorithm.

```
Field width,10,150,10
Field height,10,150,10
Corridor turn probability,0.2,0.8,0.2
```

Extract 4-A: Illustration of a configuration file for the testing framework

Each line of this file defines a single parameter for the tested algorithm and includes a start value, an end value and a step value. The framework tests every combination of these parameters in turn, generating and testing the specified number of dungeons for each combination before moving on to the next. In this particular case, this means that the framework will generate a number of dungeons for each combination of inputs illustrated in Table 4-A below, where all other parameters are left at their default values.

Field width	Field height	Corridor turn probability
10	10	0.2
10	10	0.4
10	10	0.6
10	10	0.8
10	20	0.2
10	20	0.4
10	20	0.6
10	20	0.8
10	30	0.2
…	…	…
150	150	0.8

Table 4-A: The sequence of tests carried out by the framework when it is provided with the configuration data shown in Extract 4-A (abridged)

4.2.2 System Testing Results

The framework was run using the command line prompt shown in Extract 4-B below to generate 50,000 dungeons using the digger algorithm with its default parameter settings.

```
java -jar GeneratorSystemTest.jar
  digger
  50000
  -out diggerTrial.csv
  -format csv
```

Extract 4-B: The command line prompt use to run the testing framework and produce the results discussed in this section

This populates the file "diggerTrial.csv" with 50,000 entries recording an index, the seed value of each dungeon, the percentage of empty cells in that dungeon and whether or not the dungeon is fully interconnected, as illustrated in Extract 4-C below.

```
1,8875145527919956016,0.481250,true
2,-486288242287026619,0.621250,true
3,4284796044218912235,0.476875,true
4,-3733281655860039928,0.526875,true
5,-4071258174902404846,0.468125,false
...
50000,842264251494553654,0.596250,true
```

Extract 4-C: The output file produced when testing the digger algorithm (abridged)

The benefit of saving the results in this format is that they can then be loaded in Microsoft® Excel® for further analysis by the tester. The graph shown in Figure 4-A below illustrates the interconnectivity of the 50,000 dungeons created by this algorithm.

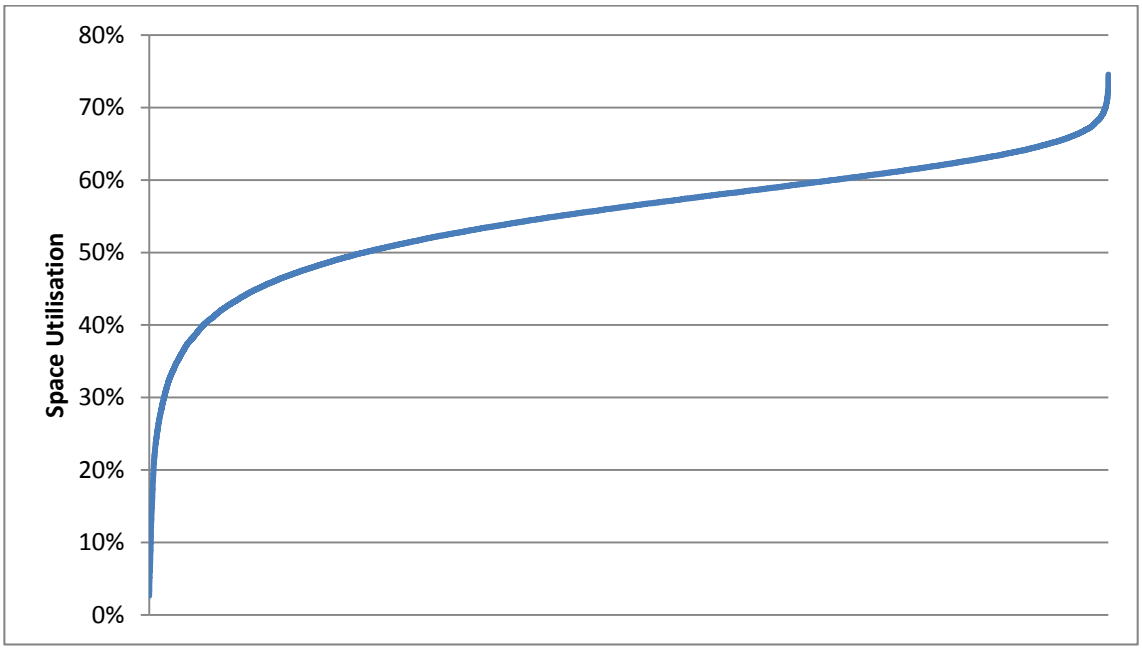

Figure 4-A: The distribution of space utilisation across 50,000 test dungeons created using the digger algorithm with its default parameter values (Microsoft® Excel® 2010)

This graph shows that the majority of the sampled dungeons produced by the digger algorithm utilised between 45% and 60% of the available space, which is considered to be acceptable for this algorithm. Additionally, 96.3% of the dungeons produced during this test were fully interconnected, providing confidence that the digger algorithm can generally be relied on to produce valid content. The trend followed by the graph above does highlight the presence of unacceptable outliers (dungeons that utilise too little or too much of the available space), but the algorithm appears to function appropriately in the majority of cases.

5 Conclusion

Overall, I feel that I was largely successful in achieving the aims and objectives set out at the start of this project. A pseudo-random dungeon generation artefact was designed and implemented in a fashion that allows it to be reused by other systems, and a computer algorithm for generating dungeon environments was designed and implemented as part of this artefact, which appears to be functionally correct. The entire artefact was tested with an external program as detailed throughout section 4, and an adequate graphical user interface to the system was implemented.

With reference to the high level goal of the project, the process of designing, implementing and testing this artefact has allowed me to practice my knowledge of object-oriented software engineering through the activities documented in this report.

5.1 Project Evaluation

This section aims to evaluate the activities in the project and what can be learnt from them.

5.1.1 Algorithm Parameter Handling

One of the most prolonging aspects of the project was determining how to handle the parameter values accepted by algorithms during the implementation phase, as documented in section 3.2.2.1. The main reason for the delay in implementing this part of the system appears to have been because it wasn't considered at the time the system was being designed, which then meant that it effectively had to be negotiated into the existing design once the implementation was underway. This meant that effort applied during the design phase was wasted as old ideas were modified.

In retrospect, recognising that the algorithms would need to be able to accept parameters during the design phase would likely have resulted in a more suitable design, resulting in less wasted time during the implementation phase and a clearer interface to the system.

For example, if the project were started again, the Algorithm classes could instead be modelled as Java Beans, meaning that they would be initialised with zero-length constructors that would set all of the parameters of the algorithm to their default values. These classes would then expose accessor and mutator methods for these parameters to allow clients to modify them. This would have allowed the factory pattern to be used, thus significantly simplifying the implementation of this part of the system.

This experience highlights the need to consider all of the main requirements of the system before starting the implementation, since neglecting such requirements results in a less efficient development process overall.

5.1.2 Overuse of Design Practices

The issue discussed in section 5.1.1 above indicates a more general criticism of how the project was handled from a software design perspective, which is that I may have been overly distracted by the need to avoid class coupling in general when considering how to handle the algorithm parameter values.

The solution that was ultimately implemented, which is illustrated in Extract 3-I of section 3.2.2.1.3, has the client look-up parameters by their names, which are provided as strings of text. Despite minimising the explicit class coupling, this solution could be viewed as dangerous because it isn't type-safe; if the string is mistyped in any way, the error won't be detected until run-time. This danger is further compounded by the fact that the artefact is designed to be used as a publicly available API and would therefore theoretically be used by programmers with little to no knowledge of the implementation of the artefact.

Contrast this to the previous solution, which was to implement a library of accessor and mutator methods in the non-abstract builder classes as illustrated in Extract 3-H. While this would have created explicit coupling between the clients and the non-abstract builder classes, this coupling could be justified because it would have resulted in a far more intuitive interface for client

programmers and a reduced chance of defects in client programs, since the Java environment would then be able to detect mistyped ‘get’ and ‘set’ methods at compile-time. This also would have made it easier to modify existing algorithms in the future, as redundant parameters could simply be deprecated rather than removed completely in order to avoid breaking client code.

Furthermore, one could argue that in the implemented solution, the clients are actually still coupled to the non-abstract builders, because if the name of a parameter changes in the builder, the references to that parameter would have to be manually updated in its clients as well. With this in mind, it now appears that it is in fact impossible to remove the coupling from this part of the system.

In my view, this highlights one of the most important lessons to be taken from this project: when dealing with larger, non-trivial software engineering tasks, it may occasionally be necessary to sacrifice some of the best practices of program design in favour of keeping the design simple. In this case, following these practices too closely seems to have rendered the system harder to use than it need be, in spite of these practices being used with the best intentions of boosting maintainability and extensibility. In the future, it may therefore be preferable to think of these practices as design guidelines that can be altered as needed, as opposed to concrete rules that must be followed in all cases.

5.1.3 Digger Algorithm Evaluation

The digger algorithm designed for this project aimed to take advantage of the intricate behaviour characteristic of multi-agent systems (see section 2.2.1). For the most part, this appears to have been successful, since the algorithm does create appropriate environments that represent dungeons consisting of rooms connected to one another via corridors. Eight exemplary dungeons generated by the algorithm using its default values are given in appendix I for context.

An interesting aspect of this dungeon generation algorithm is how diggers interact with each-other when they are moving in parallel and when they meet at intersections. Because each individual digger only observes its immediate Moore neighbourhood to avoid creating corridors that are more than two cells in width, the networks of corridors that result from the algorithm can form around small clustered groups of filled cells. Often, these filled cells appear to have a regular structure to them, such as in dungeon 5 from appendix I, repeated in Figure 5-A below, which contains two corridors permeated with pillars. This appears to be representative of agent-based simulations in general, where complex patterns can emerge from the interactions between relatively simple agents.

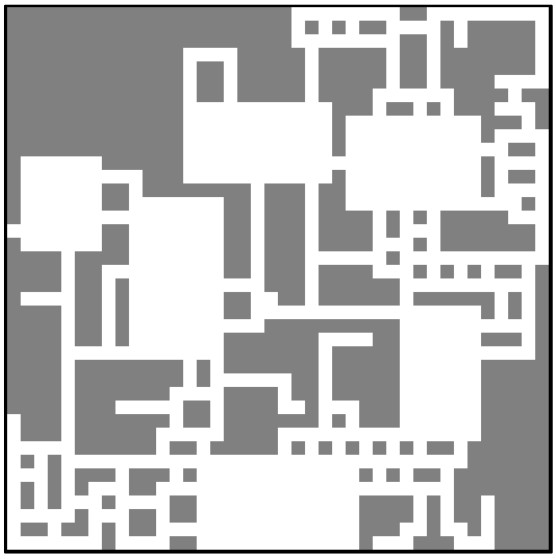

Figure 5-A: The fifth example dungeon from appendix I

One potentially unsuccessful aspect of this algorithm is that, despite efforts to avoid the formation of intersections between corridors that are more than one cell wide, such areas still

frequently emerge, such as in dungeons 2, 3, 4, 5 and 6 in appendix I. The reason for this is that the digger agents are limited to only inspecting the state of the cells immediately surrounding them according to the rules defined in section 2.2.3. A possible way of solving this, which would also simplify the algorithm, might be to instead observe the Moore neighbourhood around the cell that the digger wishes to move into. However, this feature of the agent's behaviour doesn't appear to render the dungeons unusable and could be seen as an unexpected effect of the agents interacting with one another, so it has been left in.

The design of the algorithm also means that not all of the dungeons it produces are fully interconnected. An example of a disconnected dungeon is dungeon 8 of appendix I, repeated in Figure 5-B below, where the room to the south western corner is disconnected from the rest of the dungeon.

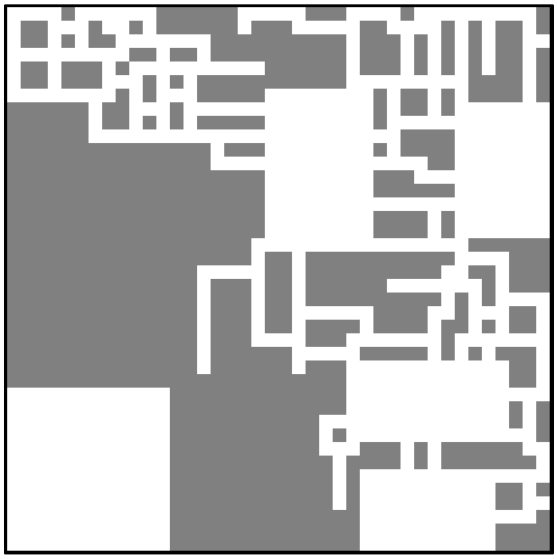

Figure 5-B: The eighth example dungeon from appendix I

This property of the algorithm means that it may be classified as a "generate-and-test" algorithm as defined in section 1.1.2, meaning that the algorithm would have to be re-run until a fully connected dungeon emerges.[14] However, as found from the results of the quantitative tests recorded in section 4.2.2, only a small proportion of the dungeons produced by this algorithm are disconnected in this way, so the computational expense of discarding unacceptable dungeons and starting again does not seem to be of major concern.

It was also previously suggested that this algorithm could be measured in terms of its worst-case time complexity and worst-case space complexity. However, the stochastic nature of the algorithm means that this would have been a highly time consuming task, so the time was instead dedicated to other tasks that were viewed as being more crucial to successful completion of the project.

Finally, section 4.2.2 shows how this algorithm can occasionally produce dungeons that utilise too much or too little of the available space. While constraints could be applied to the algorithm to reduce or avoid these situations, it's likely that applying such constraints would reduce the level of variation between the dungeons produced. This would erode the benefit of using procedural content generation, since the constraints could prevent the emergence of interesting features caused by the interactions between the digger agents such as the regular cell formations highlighted above. By not applying these constraints, the client system is also left free to determine what level of utilisation is acceptable to it.

[14] It's possible to circumvent this issue by using only one origin room. This is guaranteed to produce a fully interconnected dungeon since all of the created rooms and corridors will be connected via this origin room.

5.1.4 Time Management

To assist with time management, a Gantt chart was created before the start of the project to plan the high-level activities that would take place in the thirteen weeks starting from the beginning of the second semester. In addition, a more granular weekly timetable was created to plan each hour of a typical academic week.

5.1.4.1 Full Project Gantt Chart

With reference to the Gantt chart in appendix J, I feel that I was generally able to keep up to date with the core tasks. The original intention was to produce this report in parallel with the artefact, but instead, the report wasn't started until the start of the seventh week. I feel that this was ultimately a better decision, because it allowed more time to work on the artefact in the preceding weeks.

However, even though the tests carried out appear to have been sufficient for this project, I feel that I did not dedicate as much time to testing as might be needed for an industry-grade software system. As described in section 4.2.1, it was necessary to produce a testing framework in order to carry out system tests on the artefact, which meant less time was available for actually executing these tests (in particular, the grid search feature created in section 4.2.1.1 was never actually used). Furthermore, the implementation of the artefact suffered significant delays for the reasons discussed in section 5.1.1, reducing the time available for testing the system even further.

5.1.4.2 Weekly Timetable

The weekly timetable in appendix K plans the topic of each hour of each week, which involves not only this project but also the two taught modules that I have been taking in parallel.

While it wasn't possible to precisely follow this timetable in practice, it still assisted me in remaining committed to the project and focussed on the amount of time needed to complete it.

5.2 Potential Future Enhancements

This subsection lists ways in which this artefact could be expanded as part of future work.

A clear way in which the system could be expanded is the implementation of additional algorithms for generating dungeons, such as the space partitioning algorithm defined by Shaker et al. (2015) noted in section 1.1.2. Furthermore, despite originally being intended for dungeons, this artefact also has the potential to support maze generation algorithms, such as those described by Buck (2011) given in section 1.1.2.

On a larger scale, there is also the issue of general purpose content generators. As discussed in section 1.2, Togelius et al. (2013) indicate that there is a need for general purpose content generators that can generate multiple types of content, in order to help encourage the adoption of procedural content generation in the games industry. The generator produced in this project can be reused by any arbitrary client system that can interface with Java, but it is still only capable of producing dungeons. Therefore, another area for improvement might be to implement additional generic generators for other types of content that could then be packaged with this environment generator to build up a general purpose suite of generators for a wide array of different types of content.

Based on the findings from section 1.1.3, a particular area of interest is three dimensional level generation. This wasn't pursued in this project because of time constraints, but it does appear to be a highly intriguing and relevant topic for further study in this area. The existing coordinate representation shown in Figure 2-C could theoretically be expanded into three dimensions.

5.3 Findings and Final Comments

Procedural Content Generation (PCG) is an expansive specialist area that has been the topic of numerous academic studies in the past. The stochastic nature of the algorithms used in PCG makes them significantly more challenging to develop and test than the fully deterministic

algorithms seen in other programming domains. At the same time, however, this also increases their appeal for further investigation.

The general findings relating to software engineering that can be drawn from this project may be summarised as follows:

- Software designs should be mutable and open to refinements throughout the development process. This is illustrated in this project from how the design of the artefact continued to undergo significant changes once the implementation had started.
- For the reasons discussed in section 5.1.2, it has been made clear that the theoretical best practices in software design may not always be suitable or desirable in practice, especially in the context of non-trivial software systems, depending largely on the specific needs of the problem at hand. The practices are better treated as guidelines.
- Performing effective software testing is an intricate and time consuming process, especially when applied to larger systems. The amount of time needed to perform the required level of testing was underestimated as detailed in section 5.1.4.1. In future projects, it may therefore be wiser to dedicate more time to this task to ensure that sufficient testing can take place.

The artefact produced in this project suggests that it is possible to implement general purpose generators for non-aesthetic types of content for video games and other multimedia products. Generic “plug-and-play” generators of this nature can be integrated into existing development projects with little effort, thus contributing towards more wide-ranged acceptance of PCG as a tool for creating multimedia software more rapidly and efficiently than the current, fully manual technique allows.

6 Glossary

Accessor Method: an object method that simply returns the value of a specific field held by that object without altering its state.

API: an abbreviation of "Application Programmer's Interface".

Application Programmer's Interface: describes a library of pre-compiled object classes in an object-oriented paradigm, which are normally generic and can be repurposed for a large number of domain-specific problems.

Dungeon: a term used to refer to environments made up of rooms connected to one another using corridors.

Law of Demeter: an object-oriented design guideline stating that a given client class should only interface with its immediate suppliers and should not normally interface with the suppliers of that supplier (Lieberherr, 2003).

Middleware: refers to generic software subsystems that are used to perform specific low-level tasks on behalf of a client system. Examples of middleware components in games include physics engines and graphics engines.

Moore Neighbourhood: in a regular grid of cells, the Moore neighbourhood of a particular cell includes all of the cells that are orthogonally and diagonally adjacent to that cell within a particular radius, forming a square shaped neighbourhood (LifeWiki, 2014).

Mutator Method: an object method that accepts one or more parameters and uses these to modify the state of the object in some way.

PCG: an abbreviation of "Procedural Content Generation".

Principle of Least Knowledge: an alias for the "Law of Demeter".

Procedural Content Generation: the application of computing that involves generating interactive multimedia content using a pseudo-random computer algorithm rather than by manual design.

Pseudocode: an informal code format used to describe an algorithm as a sequence of computational steps without actually implementing it in a formal computer programming language.

Pseudo-randomness: a technique used by fully deterministic computer programs to produce the illusion of random data and behaviour. A pseudo-random sequence of numbers is produced by taking a "seed" number and using it in a complex mathematical equation that produces the next pseudo-random number and modifies the seed in a deterministic way. This means that the same sequence of numbers is produced for each seed, hence why it is termed as pseudo-random as opposed to truly random.

Von Neumann Neighbourhood: in a regular grid of cells, the von Neumann neighbourhood of a particular cell includes all of the cells that fall within a particular Manhattan distance of the origin cell, forming a diamond shaped neighbourhood (LifeWiki, 2016).

7 References

Adams, D. (2002) *Automatic Generation of Dungeons for Computer Games*. BSc. thesis. University of Sheffield.

Albahari, B. (2000) *A Comparative Overview of C#, 26. Interoperability* [Online] Available from: http://genamics.com/developer/csharp_comparative_part13.htm [Accessed: 1st April 2016].

Anderson, M. (2014) *Dungeon-Building Algorithm* [Online] Available from: http://www.roguebasin.com/index.php?title=Dungeon-Building_Algorithm&oldid=38421 [Accessed: 18th January 2016].

Buck, J. (2011) *Maze Generation: Algorithm Recap* [Online] Available from: http://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap [Accessed: 1st April 2016].

Dimovska, D., Jarnfelt, P., Selvig, S. & Yannakakis, G. N. (2010) *Towards Procedural Level Generation for Rehabilitation*.

Doull, A. (2008) *The Death of the Level Designer* [Online] Available from: http://njema.weebly.com/uploads/6/3/4/5/6345478/the_death_of_the_level_designer.pdf [Accessed: 12th December 2015].

Freeman, E., Freeman, E., Sierra, K. & Bates, B. (2004) *Head First Design Patterns*. 2nd edn. California: O'Reilly Media. pp. 243-247.

Hendrikx, M., Meijer, S., van der Velden, J. & Iosup, A. (2013) *Procedural content generation for games: A survey*, ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP), vol. 9, no. 1.

Hsu, C. W., Chang, C. C. & Lin, C. J. (2010) *A Practical Guide to Support Vector Classification* [Online] Available from: https://www.cs.sfu.ca/people/Faculty/teaching/726/spring11/svmguide.pdf [Accessed: 28th March 2016]. pp. 5-8.

Johnson, L., Yannakakis, G. N. & Togelius, J. (2010) *Cellular automata for real-time generation of infinite cave levels*.

Lethbridge, T. C. & Laganière, R. (2005) *Object-Oriented Software Engineering: Practical Software Development using UML and Java*. 2nd edn. Maidenhead: McGraw-Hill Education. pp. 371-424.

Lieberherr, K. (2003) *Law of Demeter (LoD)* [Online] Available from: http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/general-formulation.html [Accessed: 12th March 2016].

LifeWiki (2014) *Moore neighbourhood* [Online] Available from: http://www.conwaylife.com/w/index.php?title=Moore_neighbourhood&oldid=19886 [Accessed: 20th March 2016].

LifeWiki (2016) *Von Neumann neighbourhood* [Online] Available from: http://www.conwaylife.com/w/index.php?title=Von_Neumann_neighbourhood&oldid=21885 [Accessed: 20th March 2016].

Metsker, S. J. & Wake, W. C. (2006) *Design Patterns in Java*. 2nd edn. Massachusetts: Addison Wesley Professional. pp. 159-166.

Nystrom, B. (2014) *Game Programming Patterns, Chapter 8. Double Buffer* [Online] Available from: http://gameprogrammingpatterns.com/double-buffer.html [Accessed: 20th March 2016].

Procedural Content Generation Wiki (2015) *Dungeon Generation* [Online] Available from: http://pcg.wikidot.com/pcg-algorithm:dungeon-generation [Accessed: 23rd March 2016].

Procedural Content Generation Wiki (2016) *Mazes* [Online] Available from: http://pcg.wikidot.com/pcg-algorithm:maze [Accessed: 1st April 2016].

Radeck, K. (2003) *C# and Java: Comparing Programming Languages* [Online] Available from: https://msdn.microsoft.com/en-us/library/ms836794.aspx [Accessed: 1st April 2016].

Roden, T. & Parberry, I. (2004) *From Artistry to Automation: A Structured Methodology for Procedural Content Generation*.

Shaker, N., Liapis, A., Togelius, J., Lopes, R. & Bidarra, R. (2015) 'Constructive generation methods for dungeons and levels (DRAFT)'. In Shaker, N., Togelius, J. & Nelson, M. J. (eds.) *Procedural Content Generation in Games: A textbook and an overview of current research*. Location unknown: Springer. pp. 31-55.

Shiffman, D. (2012) *The Nature of Code, Chapter 7. Cellular Automata* [Online] Available from: http://natureofcode.com/book/chapter-7-cellular-automata/ [Accessed: 7th March 2016].

SpeedTree. (2016) *SpeedTree* [Online] Available from: http://www.speedtree.com/ [Accessed: 10th March 2016].

Togelius, J., Champandard, A. J., Lanzi, L., Mateas, M., Paiva, A., Preuss, M. & Stanley, K. O. (2013) 'Procedural Content Generation: Goals, Challenges and Actionable Steps'. In Lucas, S. M., Mateas, M., Preuss, M., Spronck, P. & Togelius, J. (eds.) *Artificial and Computational Intelligence in Games.* Wadern: Dagstuhl Follow-Ups. pp. 61-75.

Togelius, J., Yannakakis, G. N., Stanley, K. O. & Browne, C. (2010) *Search-based Procedural Content Generation*.

van der Linden, R., Lopes, R. & Bidarra, R. (2014) *Procedural Generation of Dungeons*, IEEE Transactions on Computational Intelligence and AI in Games, vol. 6, no. 1.

Vanegas, C. A., Kelly, T., Weber, B. Halatsch, J., Aliaga, D. G. & Müller, P. (2012) *Procedural Generation of Parcels in Urban Modelling*, vol. 31, no. 2. Massachusetts: Blackwell Publishing.

Wolfram, S. (2002) *A New Kind of Science*. Champaign: Wolfram Media. pp. 23-39. Available from: http://www.wolframscience.com/nksonline/section-2.1 [Accessed: 26th March 2016].

Zyda, M. (2005) 'From Visual Simulation to Virtual Reality to Games'. In Helal, S. (ed.) *Computer*. IEEE Computer Society, vol. 38, no. 9, pp. 25-32.