Technical Design Document (TDD)

[Introduction](#h.b1jd59vhbdtz)

[CRC cards](#h.72trjrv80ut9)

[Architecture (IMAGE LINK) (UMLET LINK)](#h.yn7g5aglwc3a)

[Approaches](#h.3ih10bax1gf0)

[Feature Design](#h.lxltp4ogdjr6)

[Setup Tasks](#h.hygj3x8q92gw)

[Task 1: Install required middleware](#h.l7dargi74jln)

[Task 2: Initial Application stub](#h.l52ci8wgn4j3)

[Movement of Player character](#h.spna9l5obie)

[Task 1: Create Game class](#h.snhl36m7cnr)

[Task 2: Create Hero](#h.qlg7orva4ls8)

[Task 3: Setup Level](#h.xo8it1wfed71)

[Hero combat](#h.se6xbdaxjg8l)

[Task 1: Create enemies.](#h.4si6ab81nakd)

[Task 2: Implement SAT collision between units](#h.8uw13r4ycprv)

[Task 3: Have hero attack enemies.](#h.g7x85dpsfzyr)

[Tower placement](#h.ic62zfh3ixlv)

[Write Tower class](#h.bi5y9ynqc7wx)

[Implement tower-terrain collision](#h.a7z3ags9upvx)

[Place tower](#h.dynaohx2ndgn)

# Introduction

A Description of each feature[[1]](#footnote-0)/user story[[2]](#footnote-1) is held in Game Design Document (GDD). The GDD describes *what* will be built.

The Technical Design Document (TDD) describes *how* it will be built - it describes the detailed design of the system and its components. Like the GDD, the TDD will grow and evolve over the lifetime of the project. But at a minimum, the TDD should contain detailed design for all the features to be implemented in the upcoming sprint.

**NB**: Before starting to write the TDD, ensure all questions are answered about *what* will be built.

The TDD includes:

* A complete set of CRC cards (these are **physical** cards)
* The architecture (UML class diagram)
* Approaches - describing different approaches that could be taken, pros and cons of the different approaches and a decision on which approach was chosen.
* Features - For each feature in the system (use the same names as the feature in the GDD), give a breakdown into a set of tasks that are clear enough for a developer to implement without asking any questions.  
  For more complex features, diagrams (such as activity diagrams and sequence diagrams) should be added to make it clearer.

# CRC cards

Example 1[[3]](#footnote-2): [link to set of photos of physical cards]

# Architecture ([IMAGE LINK](https://drive.google.com/file/d/0B5ZtMyhsDv63cExvc3JoUEFfQ0U/view?usp=sharing)) ([UMLET LINK](https://drive.google.com/file/d/0B5ZtMyhsDv63MGdlN2E5ZHplaEU/view?usp=sharing))fyp_wip.png

# Approaches

[**XML**](http://www.grinninglizard.com/tinyxml2/) **Levels:**

I’m choosing to store my levels in xml as I already have code for saving and loading to/from xml.

**Why** [**Boost**](http://www.boost.org/)**?** (...because it’s awesome!)

I’m going to use Boost’s [Thread library](http://www.boost.org/doc/libs/1_59_0/doc/html/thread.html) over the standard thread library or SFML’s thread library because not only is it recommended by the developers of SDML (over their library) but it is relatively easy to use and simple to read.

I also plan on using Boost’s [Smart Pointers](http://www.boost.org/doc/libs/1_59_0/libs/smart_ptr/smart_ptr.htm) (shared, scoped and unique pointers) to keep pointers from becoming a headache and to make the code easier to understand. The majority of stackOverflow seems to recommend using these over conventional pointers, the same way we use vector<> over conventional arrays.

I plan to utilize Boost’s [Signal2](http://www.boost.org/doc/libs/1_59_0/doc/html/signals2.html) library ([Signal](http://www.boost.org/doc/libs/1_59_0/doc/html/signals.html) is deprecated) over the [C++ Unified Event Model](https://msdn.microsoft.com/en-us/library/aa984511%28v=vs.71%29.aspx) for events as it is much more simple and is incredibly easy to use. I would argue it is easier than C# events.

For each unit needed in a level, I can load a prototype into memory and create new instances from that prototype. I can store each instance of a unit in a pool, and reuse them as necessary. The general flow would be this: request unit from pool, pool clones a new one from prototype if none available, pool returns requested unit, unit is released back to pool upon death. The following is a summary of the design patterns:

[**Prototype pattern**](https://sourcemaking.com/design_patterns/prototype)**:**

* “*Creation through delegation.*”
* Keep a prototype of each enemy in memory.
* “*Prototype co-opts one instance of a class for use as a breeder of all future instances.*”
* When we need to create a new instance of that type of enemy, we clone the prototype and avoid the *new* keyword.

[**Object Pool**](https://sourcemaking.com/design_patterns/object_pool)**:**

* Make enemies reusable.
* When an enemy dies, return it to the pool.
* When we need to create a new enemy, request one from the pool.
* Can complement the prototype pattern.

# 

# Feature Design

## Setup Tasks

### Task 1: Install required middleware

* SFML
* Boost
* Thor
* tree.hh

### Task 2: Initial Application stub

* Create application stub
* Create local repo & github repo
* Push Repo to remote

## 

## Movement of Player character

### Task 1: Create Game class

Create a Game class to hold the Hero instance. Call setDestination() on Hero whenever the mouse is clicked, pass the mouse position as the destination.

* boost:signal2 onMouseClick
  + Hook Hero’s setDestination() onto this
  + Trigger the signal when a mouse click is pressed and pass the position of the cursor
* int run() -- called from main() as “*return game.run();”*
  + Start the update and drawing thread. Each thread needs a loop but must have a different conditional bool. If one thread stops or throws an exception, ensure both threads stop their loops and join the main thread.

### Task 2: Create Hero

Create a hero class with placeholder sprite. Have him move at a constant rate toward where the player clicks.

* void setDestination(sf::Vec2 dest)
  + Hook this onto Game::onMouseClick
* void update(sf::Time elapsedTime)
  + Move the hero toward his destination
* void draw(sf::RenderTarget r)

### Task 3: Setup Level

Initialise sceneManager and level class.

* Write scene manager class and scene interface. ([see UML Diagram](#h.yn7g5aglwc3a))
* Write placeholder level class; include only elements necessary for task.
* Migrate hero from Game class to Level class.
  + Level’s update() and draw() should call the respective methods on the hero.

## 

## 

## Hero combat

### Task 1: Create enemies.

Write Pawn base class and have Hero and Minion inherit from it. Pawn keeps track of position, speed, health, etc (see diagram). Write Minion class and have all instances belong to the ENEMY faction.

Store the enemies in the level class.

### Task 2: Implement SAT collision between units

Write Collidable class and have both Minion and Hero inherit from it. Give minions and hero a round base around their feet and use that for collision.

* Write the CollisionSystem class; it needs methods to (de-)register from collision checking and, therefore, needs to keep track of which objects should be checked.
* Tick the CollisionSystem every time in the update thread.

### Task 3: Have hero attack enemies.

When the user clicks on an enemy have the hero path to the enemy and begin attacking. Have the enemy attack back.

* Hero will need to track his target, as will minions. Do this in the Pawn base class.
* Instead of adding chai script at this time, just hard code Minion behaviour.

## 

## 

## Tower placement

### Write Tower class

Write Actor class, inheriting from sf::Sprite and Collidable and derived by Pawn and Tower. Remove any redundant members from Pawn (that are now handled by Actor).

Write Projectile class and ProjecilePool class (implementing Object Pool design pattern).

Write Tower class (inheriting from Actor). Register a wide circle around the tower with the CollisionSystem, this will be used for enemy detection. When an enemy collides with this circle, make them a possible target for the tower.

* Tower will need to draw from ProjectilePool (and Projectiles will need to return themselves to the pool when they’re done).
* Use a mathematical arcing function for the movement of the projectile, so that it doesn’t beeline for it’s target.
* Call Minion::takeDamage(...) when the projectile hits.

### Implement tower-terrain collision

Write the TerrainQuadTree class. It takes a texture and subdivides until each leaf contains only one colour (one terrain type) (or leaf is sufficiently small).

Implement collision between sf::Shape and the TerrainQuadTree so that tower placement can be checked (this will also be used for pathing later).

### Place tower

When a key is pressed, show a tower under the cursor. The tower base must collide with only green(grass) tree nodes in order to be a valid location. If the tower location is valid, tint the tower green, otherwise tint it red. When the mouse is clicked, place the tower if the location is valid.

1. A feature is a unit of functionality which adds value to the user's experience playing the game. [↑](#footnote-ref-0)
2. User-stories are a concept from mainstream agile software development, and describes a specific goal they wish to achieve using the software. We prefer to use the term ‘feature’ in game development as the user’s only goal is often to “be entertained”. [↑](#footnote-ref-1)
3. No need to recreate the cards digitally in the document. As long and the photos are clear and legible [↑](#footnote-ref-2)