InfectNet

The most infectious browser-game ever!

Marianna Szabó

István Lakatos

Attila Bagossy

Brief introduction

What is it?

- Massively Multiplayer
- Code-driven (custom DSL)
- Computer Virus Themed
- Pseudo Realtime Strategy Game

Workflow

- Version Control — git
- Development Flow — nvie/gitflow

- Methodology — Kanban
- Kanban Board — Waffle

- Continuous Integration — Travis CI
- Static Analysis Tool — Codacy

Project Repositories

Parent Repository

https://github.com/infectnet/infectnet-parent

Does not contain any code, but description of the common development policies among other repositories.

Server Repository

https://github.com/infectnet/infectnet-server

Hosts the source code of the game server and the game engine.

Browser Frontend Repository

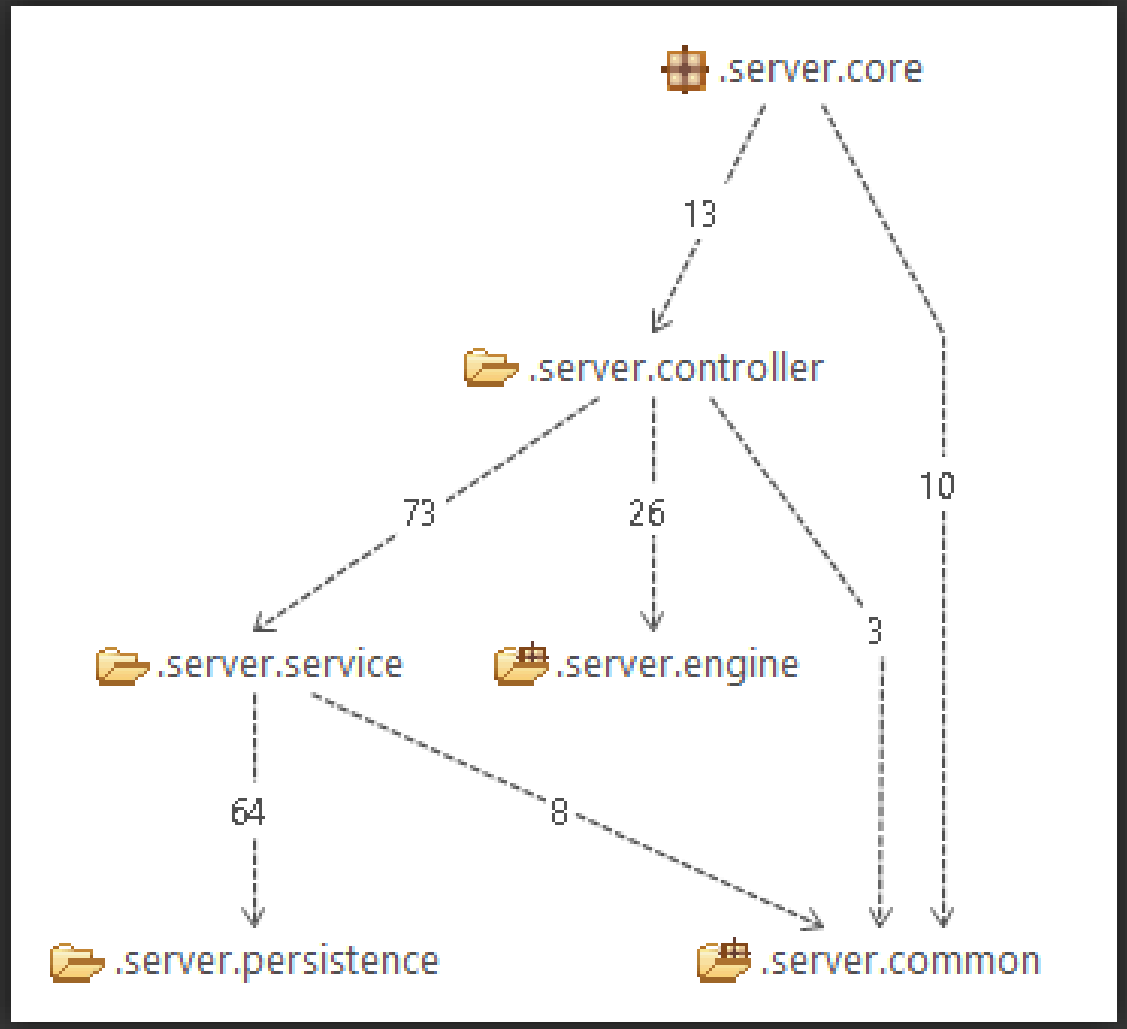
https://github.com/infectnet/infectnet-browser-frontend

Browser-based frontend for the game.

Techs that power – The server

- Build Tool — Gradle
- Web Application Framework — Spark
- Dependency Injection — Dagger 2
- Real Time Communication — WebSocket
- DSL Base Language — Groovy
- Tesing Framework — Spock

Server Structure



Engine Architecture

Complete separation of core/content

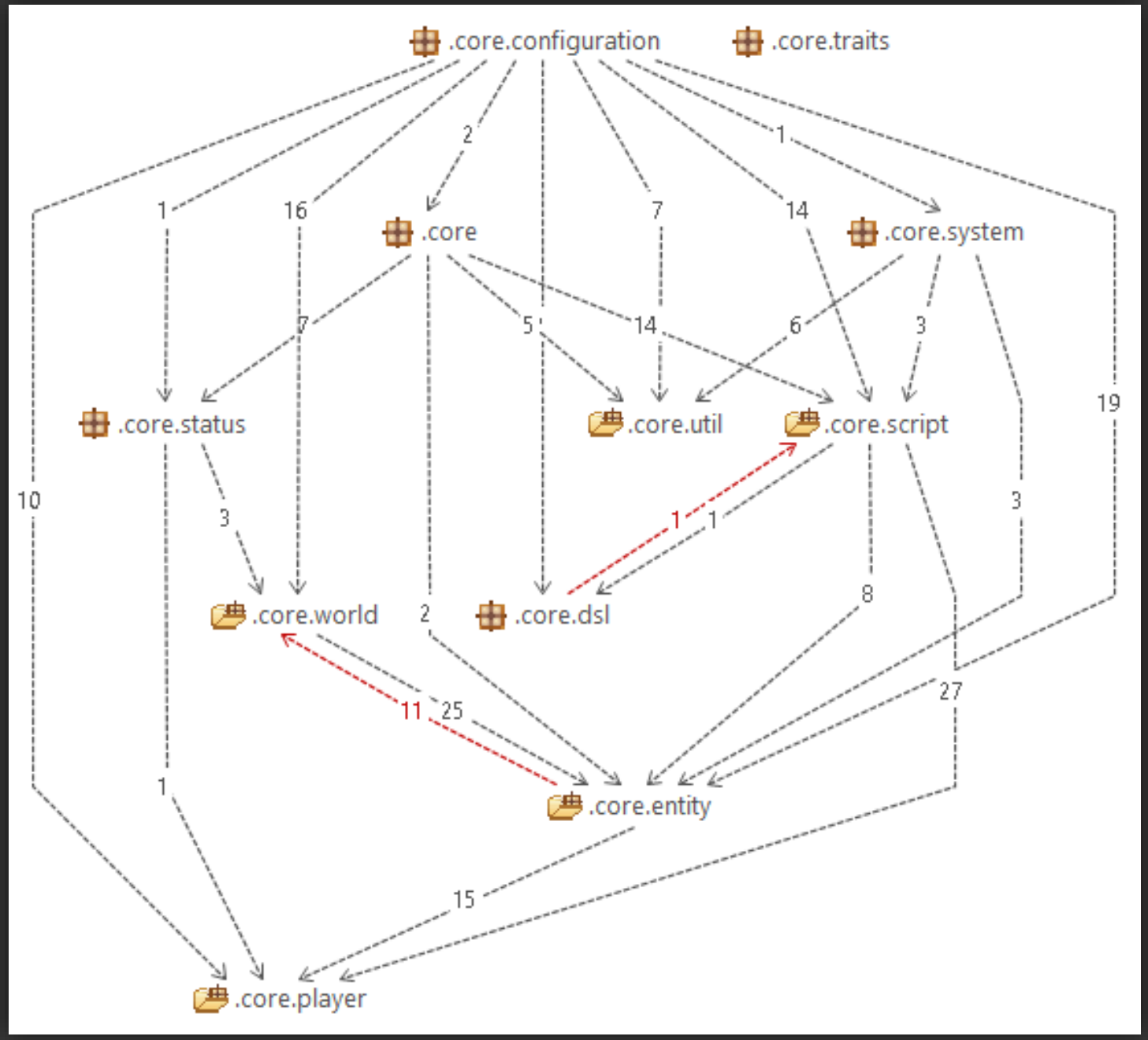
Core

- Defines interfaces to be implemented
- Self-contained, can run without content
- Provides the glue code that connects different engine parts

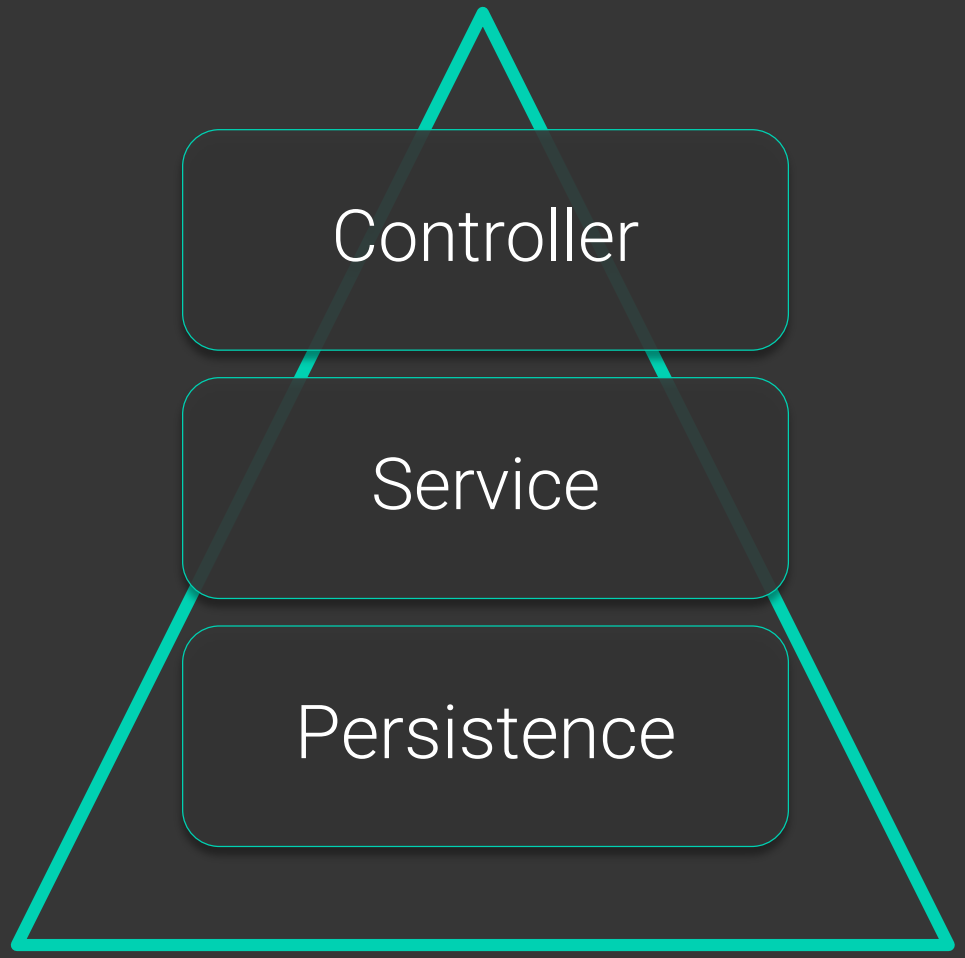
Content

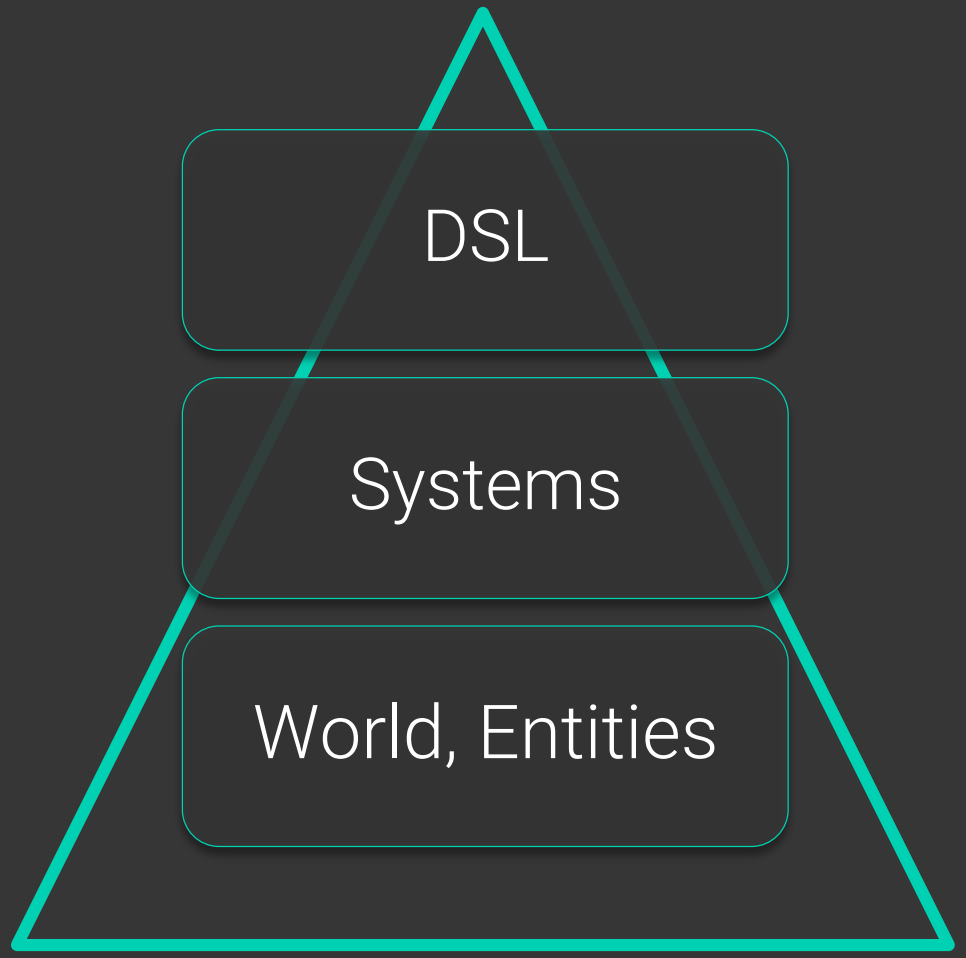
- Implementations of engine interfaces
- Must be imported into the engine
- The puzzle parts that fit into the holes of the engine

Core Package Structure



From an enterprise point of view





Domain Specific Language

Domain Specific Language

The basic idea of a domain specific language (DSL) is a computer language that's targeted to a particular kind of problem, rather than a general purpose language that's aimed at any kind of software problem.

Martin Fowler

DSL – Feature overview

- Players can express their intents in a declarative way
- Close to the natural language
- Groovy-based with our own additions

1. Select

- Select the Entities to operate on with Selectors
- Predefined Selectors are available as basic building blocks
- The players can create their own

2. Filter

- Filter the Entities by some condition
- Can include any script with any return value

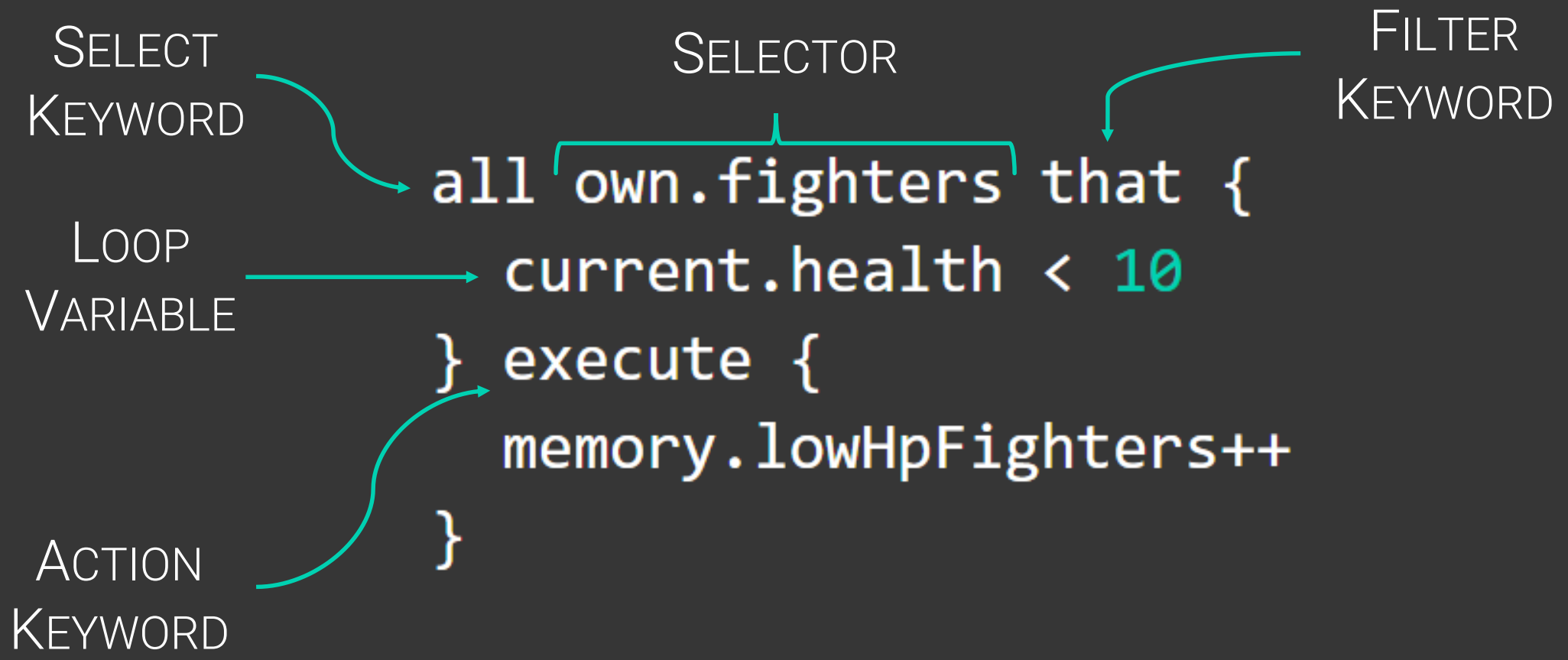
3. Action

- Perform some action on the selected Entities
- Select Entities relative to the current ones

Example SFA

```
all own.fighters that {
  current.health < 10
} execute {
  memory.lowHpFighters++
}
```

Example SFA


Select Keyword
Selector
Filter Keyword
Loop Variable
Action Keyword
all own.fighters that {
current.health < 10
} execute {
memory.lowHpFighters++
}

Example SFA – Desugared

```
this.all(this.own.getFighters()).that({
  return delegate.current.getHealth() < 10;
}).execute({
  this.memory.getAt("lowHpFighters")++;
});
```

CODE BEHIND THE ALL KEYWORD

```
public static <T> Map all(Collection<? extends T> elements) {
    return [
        that   : { Closure<Boolean> filter ->
          [execute: { Closure<Void> action ->
            doForAll(filter, action, elements);
          }]
        },
        execute: { Closure<Void> action ->
          doForAll(SelectFilterActionBlock.&trueFilter, action, elements);
        }
    ];
 }
```


SYSTEMS

Forces – Sandboxing

- Code written by the Player must be sandboxed
  - It potentially contains illegal actions
  - Directly modifying game state quickly leads to catastrophy
  - Players must not escape their own context

Forces – Determinism

- Source code execution must be deterministic
  - In the same turn each player must see the same state
  - Potentially simultaneous execution of player code must not interfere with each other

Forces – Behaviour exposure

- The DSL does not define how to do, just what to do
  - The behaviour behind the DSL objects must be put somewhere
  - A behaviour can be altered by non-local aspects

    Examples: Area-based spells, Player-level bonus, etc.

Entity Wrappers

- Entities are wrapped into an abstract proxy class called EntityWrapper
- Each entity type must have its own wrapper
- Actions can be attached to wrappers through traits

Entity Wrappers - Traits

```
@SelfType(EntityWrapper)
trait SpawnTrait {

  void spawn(String entityType) {
    this.actionConsumer.accept(this.state,
      new SpawnAction(this.wrappedEntity,
                      Objects.requireNonNull(entityType)));
  }
```

Actions (Objects)

- Command pattern
- Produced by the EntityWrappers
- Represents something that the Player wants to do
- Stored in the Action Queue (Event Queue pattern)

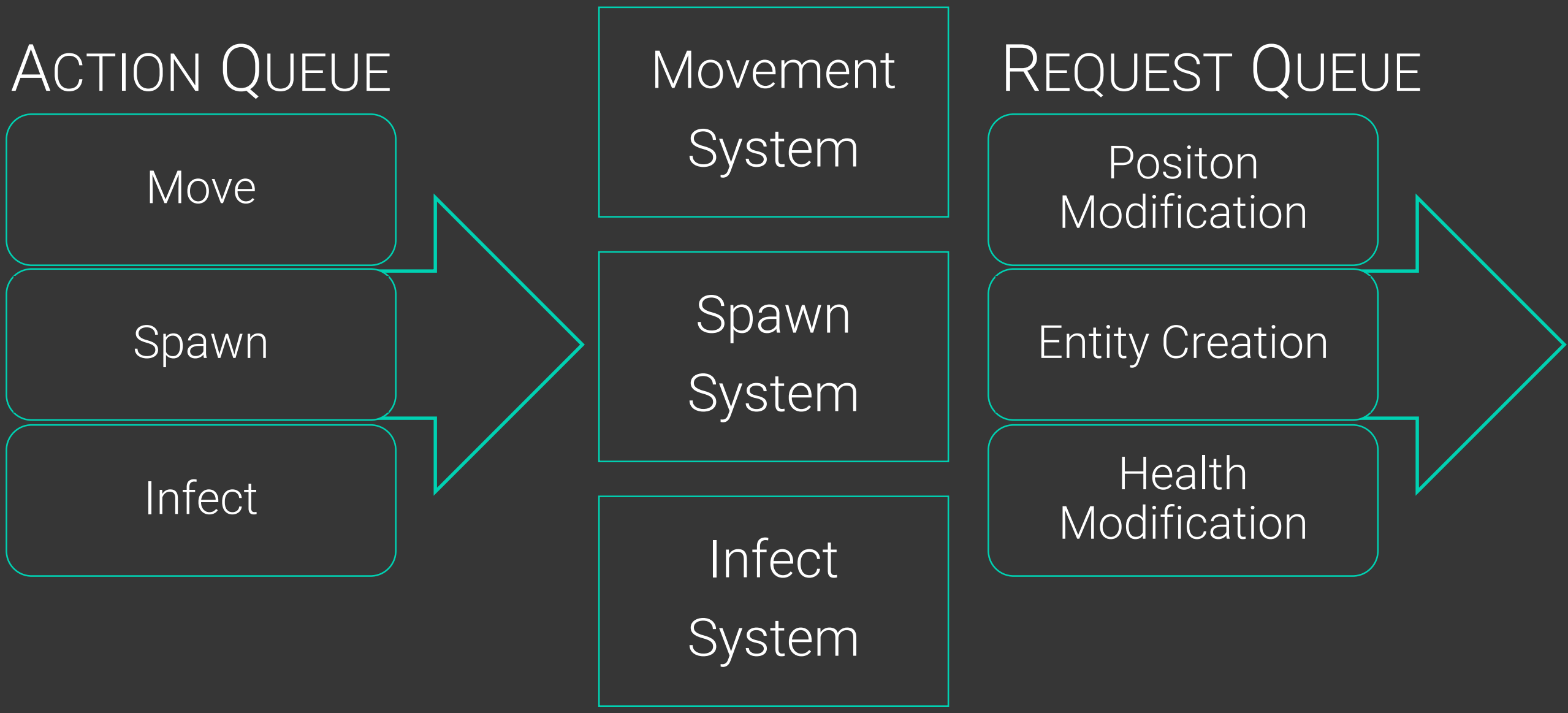
SpawnAction

```
public class SpawnAction extends Action {
  private final String entityType;

  public SpawnAction(Entity source, String entityType) {
    super(source);

    this.entityType = entityType;
  }

  public String getEntityType() {
    return entityType;
  }
}
```


Action Queue
Move
Spawn
Infect
Movement System
Spawn System
Infect System
Request Queue
Positon Modification
Entity Creation
Health Modification

Action Only System

- Observes the Action Queue
- Registers itself as a listener of specific Action types
- When triggered, processes the Action and either
  - produces a new Request which will be put in the Request Queue, or
  - Determines some factor that blocks the Action

Why do we need action and request?

Action

- DSL-level abstraction
- References the current state
  - GETS
- May fail if has illegal arguments

Request

- Atomic persistent operation
- Packs all information needed to alter the state
  - SETS
- Can be executed 99% times

Request Only System

- Observes the Request Queue
- Registers itself as a listener of specific Request types
- When triggered, executes the Request
- May put a new Request into the Request Queue

Systems

- Can listen to both Queues if they want to
- Include most of the logic that powers the game
- Can be as granular as they should be (separation of concerns)

World, Entities

Entity

- We need a common base class to handle game objects
- But game objects are very heterogenous (behaviour, attributes)
  - Let’s create an inheritance tree!
  - The tree quickly becomes unmanageable, as DIT increases

Entity

- Let's use the Component pattern instead
- Entities become component bags
- If a specific component is not necessary for an Entity, Null Object pattern can be used
- Only attributes should be stored in components, behaviour is in the Systems

Entity

```
public class Entity {
  private HealthComponent healthComponent;

  private TypeComponent typeComponent;

  private OwnerComponent ownerComponent;

  private PositionComponent positionComponent;

  ...

  private Entity() {
    /**
     * Cannot be constructed from outside, just using the Builder.
     */
  }
```

TypeComponent

- We've thrown out inheritance and have only one type
- How to create Entities with the same attributes and behaviour?
- Component pattern comes to the rescue!
- Type Object pattern

TypeComponent

- Abstract class
- The only abstract method is the one that's interesting

```
public abstract Entity createEntityOfType();
```

- It's actually a constructor method for Entities of a given type
- Basically we've created our own type system in Java

TypeComponent

- Components store the attributes, but how do TypeComponent determine the behaviour?
- TypeComponent to EntityWrapper mapping
- Handled by EntityWrapperRepository

World

- A big tilemap with two possible tile types:
  - CAVE
  - ROCK
- Backing storage can be implemented in any way
- Only implementation by now uses a two dimensional array

World

- Exposes low-level operations to the Systems

```
public List<Tile> findPath(Position start, Position target) {
  return pathFinderStrategy.findPath(this, start, target);
}

public abstract Set<Entity> seenBy(Entity entity);

public abstract Set<Entity> neighboursOf(Entity entity);

public abstract List<Tile> viewSight(Entity entity);

public abstract Tile getTileByPosition(Position position);
```

World Generation

- Procedural map generation extends the lifetime of the game
- Nobody wants to create 1000x1000 tilemaps manually
- Multiple possible strategies
  - Cellular Automaton
  - Perlin Noise
  - Voronoi Diagram based generation

Cellular Automaton

- Everybody knows: Game of Life
- Alive and dead cells correspond to CAVE and ROCK
- Multiple steps/iterations
- Can be easily adjusted through the rules and number of steps

Pathfinding

- Classic AI problem
- Performance critical part of the game
- Relaxed A* algorithm is used
  - Suboptimal path is provided
  - Runs faster than the normal A*

Game Loop

Game Loop

- An infinite loop
- Initiates player code execution and then publishes the state changes of the World and the Entities
- Uses a fixed tick time with a ScheduledExecutorService
- Reschedules itself on the Executor when finished

Game Loop Hot Path

```
private void loop() {
  Instant startTime = Instant.now();

  for (Code code : codeRepository.getAllCodes()) {
    if (code.isRunnable()) {
      scriptExecutor.execute(code.getScript().get(), code.getOwner());
    }
  }

  actionQueue.processAll();

  requestQueue.processAll();

  statusPublisher.publish(statusConsumer);

  rescheduleLoop(startTime);
}
```

Thank you for your attention!