EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

MÉDIA- ÉS OKTATÁSINFORMATIKA TANSZÉK

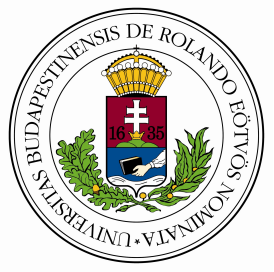
Game development in the Java programming language with the use of modern Software Engineering principles, technologies and tools

Játékfejlesztés Java programnyelven modern szoftverfejlesztési módszerek, technológiák és eszközök segítségével

Menyhárt László Gábor
műszaki tanár
matematika-fizika tanár
informatika tanár

Orosz Sándor Zoltán
programtervező informatikus BSC

Budapest, 2014



EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

MÉDIA- ÉS OKTATÁSINFORMATIKA TANSZÉK

SZAKDOLGOZAT-TÉMA BEJELENTŐ

Név: **Orosz Sándor Zoltán** EHA kód: **ORSOAAI.ELTE**

Tagozat: **nappali** Szak: **programtervező informatikus BSc**

Témavezető neve: **Menyhárt László Gábor**

munkahelyének neve és címe: **Eötvös Lóránd Tudományegyetem**

Informatika Kar

1171 Budapest Pázmány Péter sétány 1/c

beosztása és iskolai végzettsége: **műszaki tanár**

matematika-fizika tanár egyetemi diploma

informatika tanár egyetemi diploma

A dolgozat címe: **Játékfejlesztés Java programnyelven modern szoftverfejlesztési módszerek, technológiák és eszközök segítségével**

A dolgozat témája:

Szakdolgozatom tárgya egy Asteroids játék amiben egy valós fizikát idéző naprendszerben keringő aszteroidákat kell megsemmisíteni egy felhasználói bemenet által irányított űrhajóval.

A program alapja egy modern játékokban illetve játékmotorokban is használatos Entity Component System technológia egy implementációja.

A fejlesztés során az alábbi modern szoftverfejlesztési módszereket, technológiákat és eszközöket vonultatom fel:

- Fejlesztés menete: Test Driven Development
- Programnyelv: Java
- Fejlesztőkörnyezet: Eclipse
- Verziókezelés: Mercurial
- Forráskód hosztolás: Bitbucket oldala
- Taszk kezelés: Bitbucket oldalának JIRA rendszere
- Fordítás és építés. Gradle
- Folyamatos Integráció: Jenkins
- Unit tesztelés: JUnit, Hamcrest, Mockito programkönyvtárak
- Megjelenítés: OpenGL renderelés JOGL segítségével

A témavezetést vállalom:

.....

(a témavezető aláírása)

Kérem a szakdolgozat témájának jóváhagyását.

Budapest, 20…….

.....

(a hallgató aláírása)

A szakdolgozat-témát az Informatikai Kar jóváhagyta.

Budapest, 20……

.....

(témát engedélyező tanszék vezetője)

Table of Contents

Introduction

Thesis Foreword

This project demonstrates the use of modern Software Engineering principles, technologies and tools, in the development of simple two dimensional games, specifically a variation of the popular classic game called Asteroids.

Project links

The Mercurial repository of the project is hosted at:

https://bitbucket.org/FrigoCoder/asteroids/

The Continuous Integration server is found at:

http://frigo.noip.me/jenkins/job/asteroids/

An experimental demo Applet can be found at:

http://frigo.noip.me/asteroids/

Goal of the project

The goal of the project is three-fold: apart from the obvious end result that is a playable Asteroids game, the project also serves as an introductory experiment in the world of game programming, specifically game programming based on Entity Component Systems, and an example of modern software development for other developers to follow.

Target audience

Players

Developers

Software Engineering principles and technologies used

- Object Oriented Programming (OOP)
- Integrated Development Environments (IDEs)
- Software Testing
- Clean Code
- Test Driven Development (TDD)
- Mock Objects
- Distributed Revision Control
- Build Automation Software
- Continuous Integration (CI)
- Software Quality Control
- Issue Tracking System
- Entity Component Systems (ECSs)

Software Engineering tools used

- Java
- Eclipse
- JUnit
- Mockito
- Mercurial
- Gradle
- Jenkins
- SonarQube
- JIRA

Licensing

Acknowledgements

I would like to thank my friend András Dudás for sharing his expertise and insights regarding game development, with a special emphases on architecture and agent systems and for beta testing Asteroids and giving valuable programmer and user feedback.

I would like to thank my friend István Fazekas for beta testing Asteroids and giving valueable user feedback.

I would like to thank my first Scrum Master and mentor, Péter Sipos for introducing me to the world of Clean Code, Test Driven Development and Software Craftmanship

I would like to thank my current and past coworkers including Sándor Rideg, Péter Mérey, Carlos Hernandez Matas, Imre Városi, Zoltán Fermann and Tamás Pallos for sharing their knowledge, expertise and practices with me throughout the years and for being excellent team members and coworkers.

I would like to thank all of my teachers throughout the years for sharing their knowledge and perspective regarding a wide variety of subjects.

I would like to thank my supervisor, László Gábor Menyhárt for making this thesis possible.

User Documentation

Backstory

In the year 2101 AD, British scientists detected a huge wave of asteroids approaching the solar system. Their origins unknown, their danger unprecedented.

Humanity, seeing they had no chance to fight the threat without endangering Earth itself, in their last desperate attempt, constructed Arthur's Mantle. A state-of-the-art transphasic device, powered by enormous amounts of energy, to hide the Earth from the rest of the universe. This state however can not last forever - the device consumes huge amounts of energy that Earth simply can not provide in the long term.

The Great Ship was also constructed. A spaceship of epic proportions to fight the threat directly. Equipped with Orion drive as the main propulsion system and armed with a nearly unlimited amount of BFM-9001 missiles, it represents the best technology Humanity could offer.

Still, these solutions are not without a price. The cost of keeping these two projects afloat are massive. The majority of resources, workforce and technological progress of Earth are diverted to these endeavors of utmost importance without regard to anything else. Needless to say, recessions, poverty, crime and chaos is rampant.

You have been selected as Captain of this ship. You are entrusted with this task of the highest importance. You must defend our planet from this threat at all cost. We demand you do this job to the best of your abilities. We expect you to surpass them.

The existence of our civilization - or what is left of it - depends on your actions. Do not waste any time or resources in pursuit of this goal. No failure is permitted or tolerated.

System Requirements

The software requires an OpenGL 2.0 compatible video card with installed drivers for proper display capabilities and a keyboard for player input. These requirements are most likely fulfilled by current Personal Computers. Unfortunately the game in its current state does not support Android or iOS based mobile devices.

Since the project is based on Java and multiplatform libraries, it should have no problem with running on a wide range of operating systems like Windows or Linux. However only Windows and Linux is supported officially at the moment.

The computer should have a Java 1.7 Runtime Environment installed. This is usually available on most computers, but in case it is not installed or is outdated, please download and install it from http://java.com

Installation

From the webpage

Download the latest release from:

http://frigo.noip.me/jenkins/job/asteroids/lastSuccessfulBuild/artifact/build/distributions

Extract the zip file to a directory.

From the bundled CD

No installation is required, the CD copy is ready to run.

Running

After installation, run:

bin/asteroids.bat on Windows

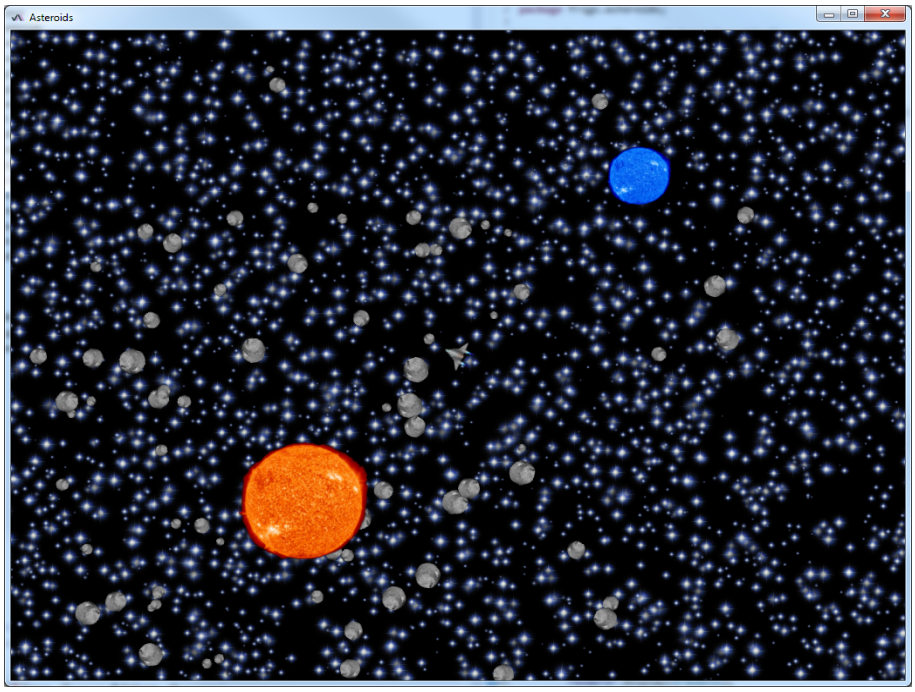
bin/asteroids on Linux

The game executable will initialize and enter straight into a new game.

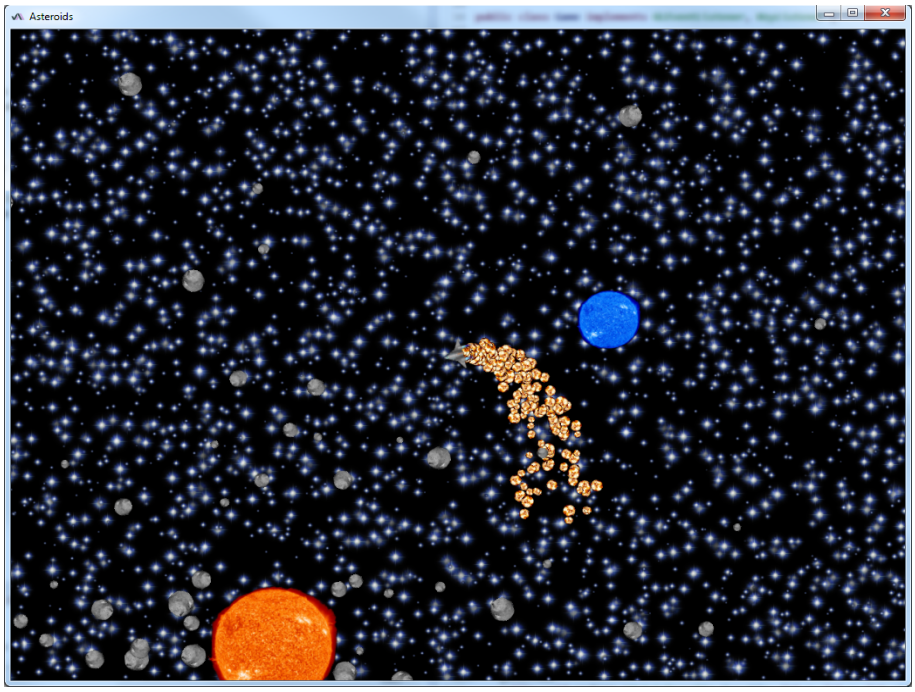
Basics

The player controls a spaceship tasked with clearing the asteroid debris in a solar system. The game uses realistic two dimensional physics. The player can only control the movement of the spaceship in a restricted manner. Forward thrust is achieved by the forward arrow on the keyboard. Angular acceleration, rather than angular velocity, is influenced by the left and right arrows. The spaceship can also shoot bullets, in order to achieve this, the player has to press the space button. The player should spend some time familiarizing himself with the controls, the realistic physics makes the control non-trivial, especially in the presence of gravitational wells.

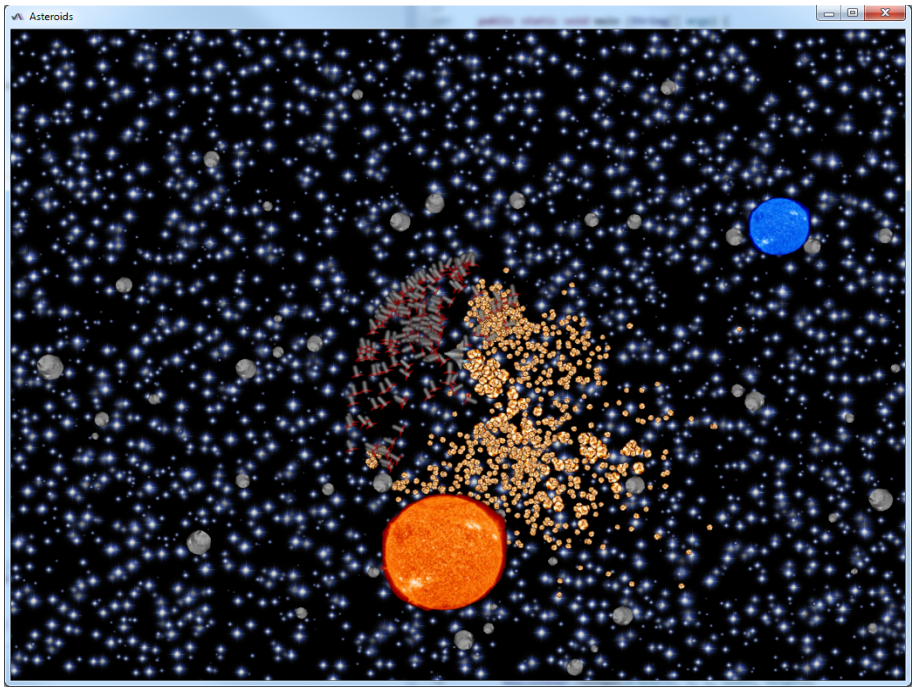
Screenshots



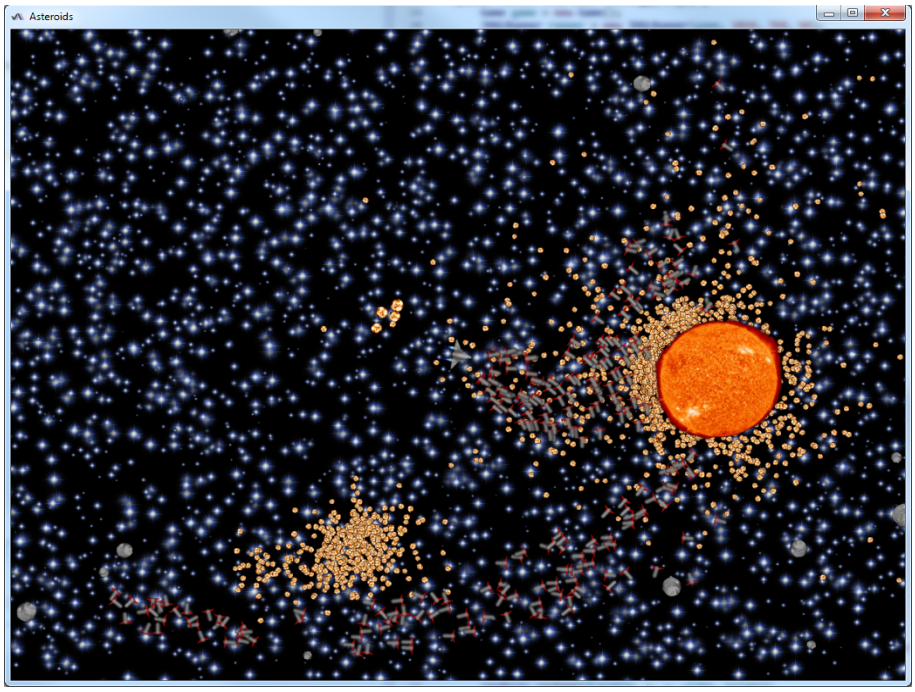
Initial screen



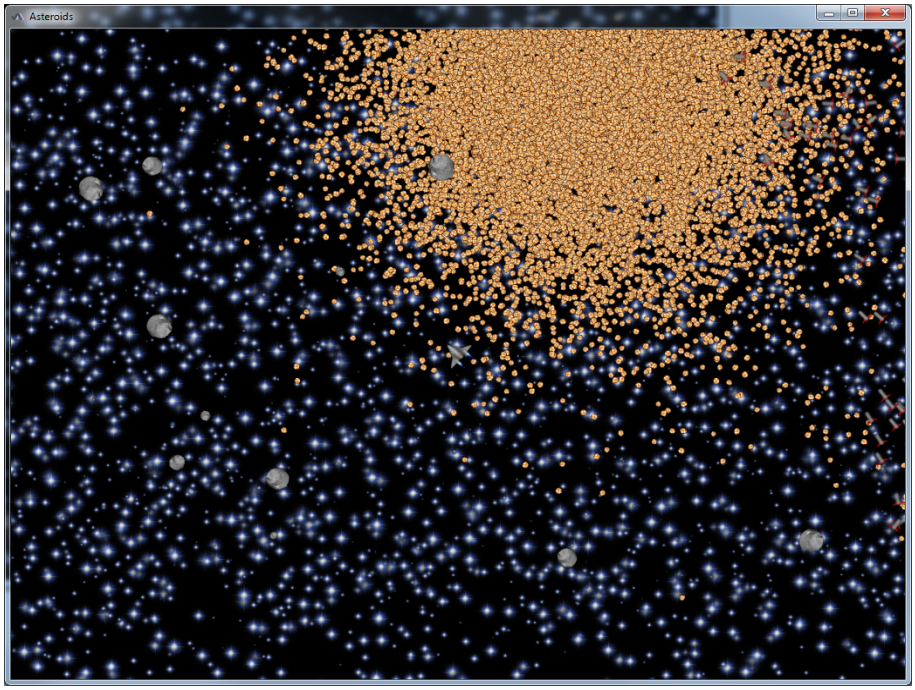
Testing the thrusters. Press the Up arrow to accelerate forward, the Left and Right arrows to accelerate the angular momentum left and right, respectively.



Testing the BFM-9001 missile system. Press and hold Space to fire.



Shooting at the Sun. You can do this by settling the starship into orbit around the sun, matching the angular velocity so you always face the sun, and shooting BFM-9001 missiles continuously.



Blowing up the Sun.

Developer Documentation

Software Engineering principles, technologies and tools used

Object Oriented Programming

Objects

Object Oriented Programming is a programming paradigm built around the concepts of objects containing fields and having associated methods. These fields describe the state of the object, and methods operate on this state in a restricted or predefined manner.

For example, an object representing a two dimensional vector might contain the *x* and *y* components of the vector, with appropriate methods that translate, scale, rotate that vector, calculate its norm, multiply a matrix with it, or calculate the dot and cross products of two vectors.

Classes

Classes, also called types, are templates for creating objects. They describe the fields that are needed for the creation of objects, the constructors that can be called to construct objects, and the methods that operate on objects of this particular type. Objects are *instances* of classes. Classes can also be thought as the categorization of objects according to some semantic meaning.

For example, a class representing a person might look like this:

```
public class Person {

    private int age;
    private Name name;
    private Address address;

    public Person (int age, Name name, Address address) {
        this.age = age;
        this.name = name;
        this.address = address;
```

```
    }

    public int getAge () {
        return age;
    }

    public String getGivenName () {
        return name.givenName;
    }

    public String getFamilyName () {
        return name.familyName;
    }

    public Address getAddress () {
        return address;
    }

    public void setAddress (Address newAddress) {
        address = newAddress;
    }

}
```

Encapsulation

Fields and methods related to the same semantic meaning should be bundled into the single class.

For example, a *Person* class should contain fields and methods related to persons, while an *Address* class should contain the specifics of addresses, like city, street, house number and ZIP code. A *Person* class should not contain these latter information directly, only indirectly through an *Address* field.

Information hiding

Encapsulation also implies information hiding. Fields and methods in the class can be hidden from the outside world. They might contain implementation specific details that would be inadvisable to reveal to the outside world, or they could assume additional constraints placed on the state of the object that the outside world would be unable to respect.

For example, and address should always require a ZIP code and therefore disable any way for the outside world to remove it from the address object.

A two dimensional vector represented by its angle and length should make sure that a null vector is always represented by a zero angle and zero length, not arbitrary angle and zero length, to prevent any possible inaccuracies in its calculations.

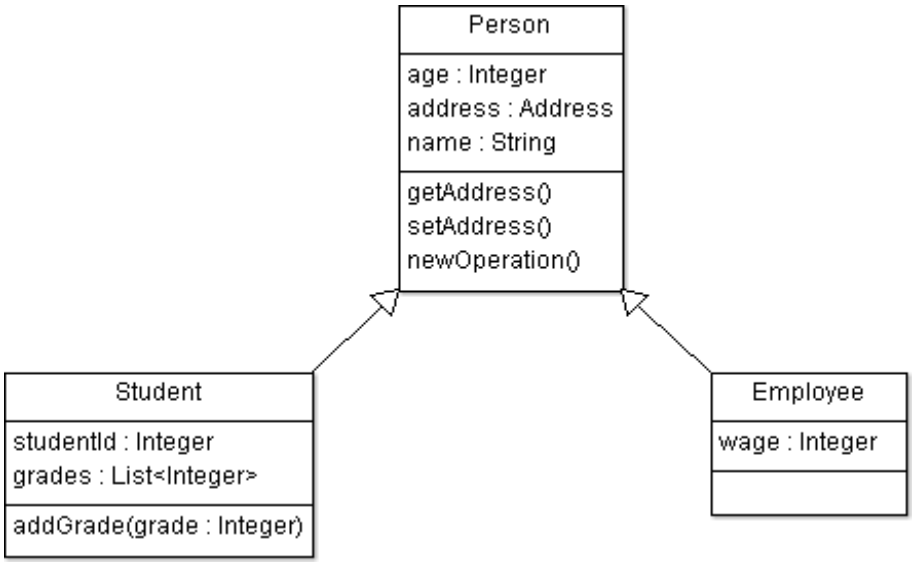
In the Person example above, all fields are private and can be manipulated only via public methods, and currently there are no methods to set a person's age or name.

The essence of encapsulation and information hiding is to provide a controlled way to modify the object's state, preserving its constraints.

Inheritance

Classes can extend or "subtype" other classes to reuse fields and methods. Subclasses can introduce new fields, new methods or override already existing methods with extended behavior.

For example, there could be a class named Student subclassing Person that includes additional fields about the student: student ID, school(s) of choice, grading information, and methods to grade him.



Inheritance example

Polymorphism

A subclass method can have different behavior than the same method in the superclass. The appropriate method to be executed is selected at runtime based on the

dynamic type or class of the object.

For example, in the case of a parent class called Animal and subclasses Cat and Dog, a method called talk() that returns a String might return “Meow!” in the case of a cat object, or “Woof!” in the case of a dog object. A method accepting an Animal object might receive either result of the method based on the actual object given to it.

Liskov Substition Principle

Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T

In other words, instances of a subclass should be substitutable into any method that expects an object of the superclass without semantically altering the function of the program.

Interfaces

Interfaces are collections of method signatures with implied semantic meaning.

A class can implement multiple interfaces, and must provide an implementation for each of these method signatures.

Example interface definition:

```
public interface Animal {
    void eat ();
    void travel ();
}
```

State

The state of an object encompasses all of the (externally observable) data contained in the object. Externally non-observable states, for example buffers in the objects, are not part of the state.

Immutable objects

Objects with fixed state. Methods return new objects rather than modifying inner state. Immutable objects are massively preferred, especially in multithreaded code.

For example, String class in Java.

Identity

The unique identity of an object distinguishes it from other objects.

For example, the memory address where the object is stored can serve as such an identity.

The Java Programming Language

The Java programming language is an Object Oriented Programming language developed by James Gosling and coworkers at Sun Microsystems and released in 1995.

Bytecode

Java is designed to be highly platform independent. Rather than compiling source code into native binary code of a given target platform, Java compilers create platform independent *bytecode* in the form of *class* files.

Java Virtual Machine

These bytecode or class files then later can be interpreted or executed by any Java Virtual Machine available on a given target platform. “Write once, run anywhere” is the slogan used by Java to illustrate this flexibility.

Bytecode is not restricted to the Java programming language, other languages also compile into bytecode that can be later run on JVMs. Clojure, Groovy, Scala are main examples of programming languages designed for JVMs.

Java Runtime Environment

Java Runtime Environments bundle a JVM and Java Class Libraries, standard libraries available for use by Java programs on all target platforms. A JRE is required on a target platform to run compliant Java programs.

Syntax and features

Object Oriented nature

Java derives most of its syntax from C and C++ but aims to provide higher level language constructs and better support for Object Oriented Programming. Java enforces the use of modularity in the form of *packages*, *classes* and *visibility levels*.

Superclasses and interfaces

Java supports single inheritance from a superclass and implementation of several interfaces as opposed to the multiple inheritance support of C++.

Threads

Java provides threads and concurrency features as a standard language feature.

Robustness and security

Java gives precedence to robustness and security over performance. It especially eschews *undefined behavior* of operations under boundary conditions that is so prevalent in C++ due to unavoidable differences in target platforms.

Exceptions

As part of the move towards robustness and security, Java uses *exceptions* instead of undefined behavior or special return values to signal exceptional cases or errors.

Type system

Java data types are broken down into three categories: *primitives*, arrays and objects, or instances of classes. Primitives can be *boolean*, *byte*, *short*, *int*, *long*, *float* or *double*. Arrays can be arrays of primitives, arrays or objects.

Java passes primitive parameters by value, and array and object parameters by reference. This is in contrast of the confusing nature of passing by value, pointer, reference, or recently, by rvalue reference in C++.

Java only supports signed integer primitives. Intermixing and conversion of unsigned and signed values is a common source of error in C++.

Memory management

Java also avoids the use of manual memory management that is also a common source of error in C++ software, offering *Garbage Collection* as an alternative.

Generics

Java does not support specialized template types. Rather, it supports the use of *generics*, a syntax sugar for more convenient handling of parameterized collections and lists.

Compiler support

Java favors simplicity and elegance of the language rather than the convenience of writing a compiler for it. Yet, because of the enforced modularity and lack of template type system, compilation of Java sources are generally faster than C++ sources.

Applet

Applets are special Java programs that can be embedded in web pages or other applications. They run in a restricted security sandbox. Their use gradually fell out of favor because of the common security vulnerabilities found in JREs and the inconvenience presented to end users.

Servlet

Servlets are server side HTTP request handlers. They allow web servers to be written in Java.

Examples

These are by no means comprehensive examples for Java applications and classes. The demonstration of the full capabilities of Java are out of the scope of this thesis.

Hello World example

This trivial Hello World example shows the use of package declarations, static main methods, constructors, private fields and the use of a standard Java method `println` of the `System.out` static final field.

```
package frigo;
```

```
public class HelloWorld {

    public static void main (String[] args) {
        HelloWorld hello = new HelloWorld("Hello World!");
        hello.greet();
    }

    private String whatToSay;

    public HelloWorld (String whatToSay) {
        this.whatToSay = whatToSay;
    }

    public void greet () {
        System.out.println(whatToSay);
    }

}
```

Fibonacci example

This example shows interface implementation and the use of `HashMap`, a standard container in Java for the purposes of memoization.

```
package frigo.asteroids;

import java.util.HashMap;

public class FibonacciCalculator implements FibonacciCalculatorIf {

    private HashMap<Integer, Integer> buffer = new HashMap<>();

    public FibonacciCalculator () {
        buffer.put(0, 1);
        buffer.put(1, 1);
    }
```

```
    @Override
    public int getFibonacciNumber (int index) {
        if( buffer.containsKey(index) ){
            return buffer.get(index);
        }
        int result = getFibonacciNumber(index - 2) + getFibonacciNumber(index
- 1);
        buffer.put(index, result);
        return result;
    }

}
```

```
package frigo.asteroids;

public interface FibonacciCalculatorIf {

    int getFibonacciNumber (int index);

}
```

Car example

This example demonstrates the use of inheritance to define car types with different behaviors, the protected visibility level and comments.

```
package frigo;

public class Car {

    protected int speed;

    public void start () {
        speed = 30;
    }
```

```
    public void stop () {
        speed = 0;
    }

}
```

```
package frigo;

public class Ferrari extends Car {

    @Override
    public void start () {
        speed = 100;
    }

}
```

```
package frigo;

public class Trabant extends Car {

    @Override
    public void start () {
        // do nothing
    }

}
```

```
package frigo;

public class CarWithNoBreaks extends Car {

    @Override
    public void stop () {
        // do nothing
    }
```

```
}
```

Integrated Development Environments

Integrated Development Environments, or IDEs for short, are software applications with graphical user interfaces that provide integrated tools for source code editing, syntax highlighting, compiling, on-the-fly syntax checking, debugging, testing, software quality, collaborative functions, and other software engineering tasks. One of their main advantages is the ease and speed of the usage of their integrated tools that can lead to faster feedback time during development and this higher throughput of better quality software.

For Java, the most widespread IDEs are:

- Eclipse
- IntelliJ IDEA
- NetBeans

Eclipse

For the writing of this thesis, Eclipse was chosen because of its widespread acceptance and the author's familiarity with it.

Eclipse is highly modular and customizable, can be adapted to a wide variety of software engineering tasks and beyond. Eclipse editions or software built on Eclipse libraries are available for Java, Java Enterprise Edition, C++, Python, Android, SQL, RSS and other programming languages and technologies.

Views

Its graphical user interface is mainly build around *views*. Views handle one aspect or technology associated with the project. Some examples include:

- Package Explorer view: Displays the structure of a Java project in a hierarchical manner, from directories to packages, classes and even members. It is of course highly customizable on what exactly to display
- Problems view: Displays the warnings and errors currently present in the project. Its scope can be restricted on what to display, Project or Selection, Warnings or

Errors criterias for example.

- Outline view: Displays members of the current class. Customizable to display only public members, methods, nonlocal types.
- Progress view: Lists the background tasks currently running in Eclipse. This can range from refreshing the workspace, compiling the project, updating the project via a Revision Control System, or running an application.
- Console view: Displays the output of the application or tests.
- JUnit view: Displays the result of the lastest run of JUnit tests.
- Search view: Displays the occurences of the last searched keyword.
- Call Hierarchy view: Displays callers and called methods of a method.
- Type Hierarchy view: Displays the (sub)type or supertype hierarchy of a class, including interfaces, subclasses and anonymous classes.
- Coverage: Displays result of various Code Coverage plugins: JaCoCo, EclEmma or Cobertura

Editors

Editors are the main windows Eclipse provides for source code editing. Depending on the content of the file being edited, their features can include:

- Tabs
- Highly customizable syntax highlighting
- Diff/merge
- Customizable annotations
- Spell checking
- Search and replace
- Code Clean Up
- Formatters

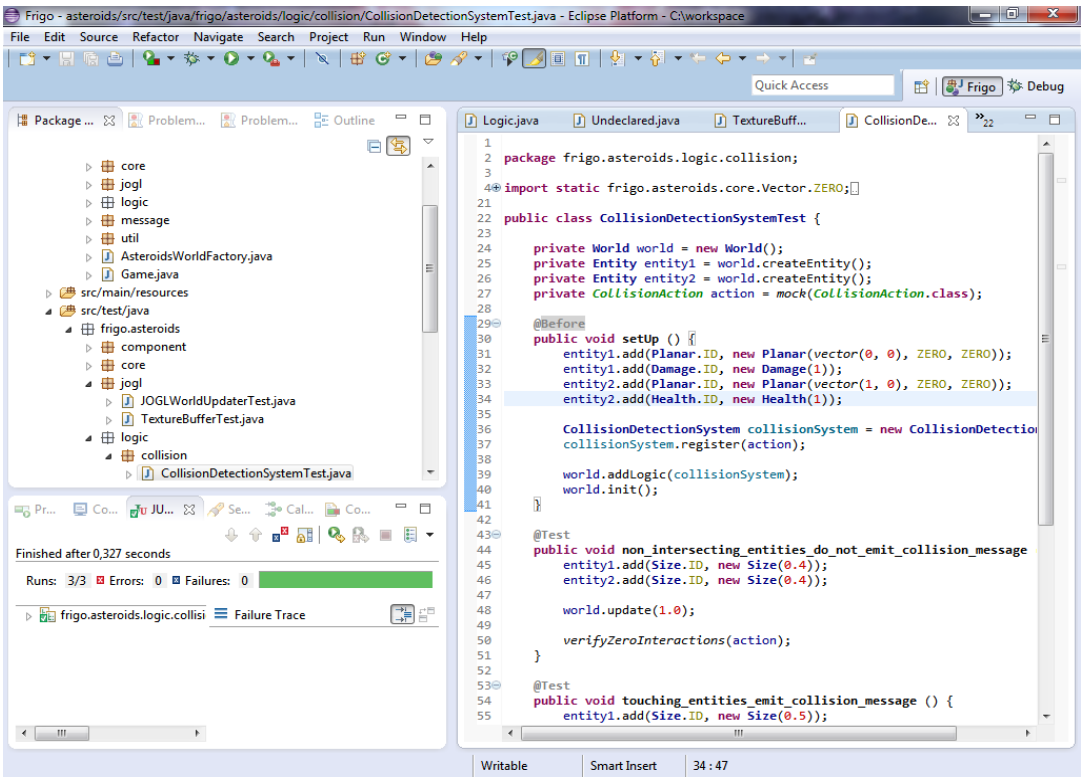
- Save Actions
- Interactive warnings and errors with customizable severity levels
- Javadoc formatting and syntax highlighting
- Various Content Assist tools:
  - Automatic insertions of class names, braces, quotes or other characters
  - Folding
  - Occurence highlighting
  - Syntax coloring
- Local Histories

Other tools

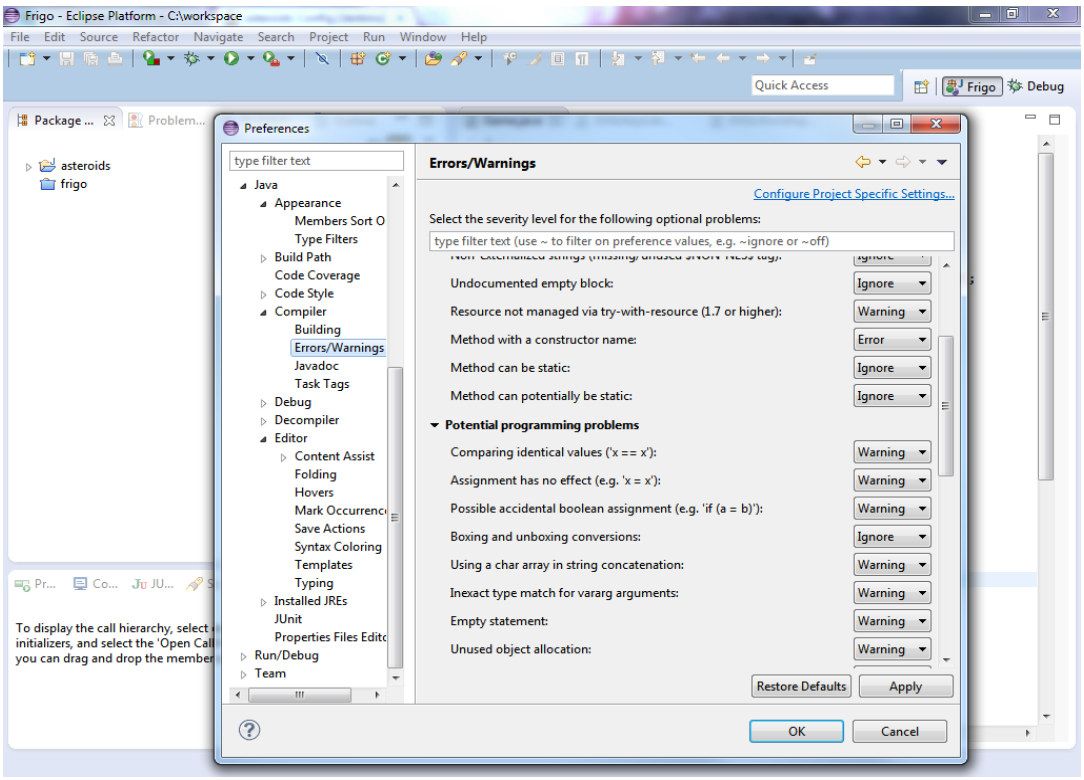
Furthermore, Eclipse has native or plugin based support for virtually all development related tasks. Examples include but are not limited to:

- Support for different perspectives to support different phases of development. For example there is a separate Debug perspective with debugging related views and toolbars, and there is a Team Synchronizing perspective for teamwork related tasks.
- Version Control Systems:
  - Concurrent Versions System (CVS)
  - Subversion (SVN)
  - Mercurial (Hg) plugin
  - Git plugin
  - ClearCase plugin
- Ant scripts
- Java Classpaths and Libraries

- Debugging features
- Different run and debug configurations



Eclipse displaying the Package Explorer view, JUnit view and a Java Editor with a JUnit test class.



Preferences page showing the numerous options for customization.

Automated Software Testing

Test

Automated Software Testing is the use of different levels of automated *tests* or *test suites*, specially written pieces of software, to verify the state and behavior of the *system under test*, which can be the application itself, its modules or its constituent units.

Test fixture

A test fixture is an environment designed to mimic the behavior of the production environment and collaborators of the system under test in some predefined manner.

Tests place the system under test in a test fixture, and verify whether it interacts with this test fixture and its collaborators in a correct manner.

For example, a disk partition management software should not be tested on real hard drives, but on mock classes of the interface representing hard drive access. The test would create these hard drive access interface mocks, create the class responsible for partition management, inject the mocks into it, execute it, and verify whether the appropriate methods are called with the appropriate parameters.

Four phases of tests

1) Setting up the test fixture: Setting up and creation of all real and fake collaborators, instantiation of the system under test and injecting the collaborators as dependencies.
2) Exercise: Interaction with the system under test in some predefined, controlled manner.
3) Verification: Verification of the state or behavior of the system under test or the effect it had on its collaborators. Less commonly, verification that a certain side effect did not occur.
4) Tearing down the test fixture: Stopping processes, closing resources, cleaning up after the setup and exercise phase.

Setup, exercise and verification are also known as given / when / then.

Language of tests

Tests are usually written in the same programming language as the application, but this is not a requirement. As long as there are appropriate interfaces for the modules or the application, it is possible to test it from other programming languages. In fact, for API tests this might be preferable, since it prevents the test from accessing module or application internals, thus assuring that it truly tests the module or application through its public interface.

Full automation

Tests should be fully automated and require no manual interaction or interpretation. Test runs should result in either success or failure, they must not rely on a user going through logs to determine whether a test run was correct.

Tests should be clearly identified as tests and grouped together in a separate module, directory or suite. Creation, addition, execution and removal of tests should be as unhindered and fast as possible.

Orthogonality

Tests should be semantically independent, that is, tests should not verify assertions that other tests already checked, or in other words, tests should be *orthogonal*.

Interference

Tests should also not influence or interfere with each other. *Test interference* is very difficult to debug. Test fixtures should restore all state that could have been modified during the test run.

Test pyramid

Tests are categorized into different levels of a *test or testing pyrami*d based on the their scope, or how much of the System Under Test they are exercisizing.

The higher a test is on this pyramid, the slower, more expensive and fragile it is. A

successful testing strategy should take this into account, and should have fewer but more efficacious tests on higher levels of the testing pyramid.

Levels of the test pyramid

GUI Tests

Their role is to test the entire application through its Graphical User Interface.

They are exceptionally slow and fragile, or sensitive to changes to the application, and thus their use should be limited to cases that lower levels of the pyramid are unable to test.

Ideally, the Graphical User Interface should be only a thin layer between the user and the Application Programming Interface and should require minimal GUI testing.

API Tests

Their role is to test the application through its Application Programming Interface via abstract application- or domain-specific concepts.

For example, an API test for a multiplayer game could connect to a game server under test, perform game actions and verify that the server sends back proper responses.

Integration Tests

Their role is to test whether two or more modules are integrated properly, or whether their interfaces are connected properly, are compatible, and they can work in harmony.

Component Tests

Their role is to test whether a component or module works in isolation according to expectations.

Unit Tests

Their role is to test individual units of source code in isolation. In Java, units are implied to be classes. These are most basic and important of automated tests.

JUnit

JUnit is a software library for Java based unit tests. It is widely supported by virtually all Java Integrated Development Environments, Build Automation Systems, Continuous Integration Software and other tools.

As of JUnit 4, test fixtures and unit tests are placed in plain java classes and methods and are distinguished by annotations.

Test classes are recreated, and test fixtures are set up and torn down for all unit tests methods.

JUnit supports Hamcrest matchers, and works exceptionally well in conjuction with Mockito, thought it is independent from it.

JUnit annotations

- `@Before` is used to annotate methods that should run as part of setting up the test fixture.
- `@After` is used to annotate methods that should run as part of tearing down the test fixture.
- `@Test` is used to annotate unit tests represented by methods.
- `@Ignore` is used to annotate unit tests that should be ignored during test runs. This is a preferred solution over commenting out unit tests.
- `@BeforeClass` is used to annotate static methods that should run once before the unit tests in the class, as opposed to `@Before` running before every method.
- `@AfterClass` is used to annotate static methods that should run once after the unit tests in the class, as opposed to `@After` running after every method.

JUnit example

`AdderTest` unit test

```
package frigo.dojo;
```

```
import static org.hamcrest.Matchers.is;
import static org.junit.Assert.assertThat;

import org.junit.Before;
import org.junit.Test;

public class AdderTest {

    private Adder adder;

    @Before
    public void setUp () {
        adder = new Adder();
    }

    @Test
    public void result_is_zero_at_start () {
        assertThat(adder.result(), is(0));
    }

    @Test
    public void result_after_one_addition () {
        adder.add(3);
        assertThat(adder.result(), is(3));
    }

    @Test
    public void result_after_two_additions () {
        adder.add(3);
        adder.add(2);
        assertThat(adder.result(), is(5));
    }

    @Test
    public void result_after_three_additions () {
        adder.add(3);
        adder.add(2);
```

```
        adder.add(1);
        assertThat(adder.result(), is(6));
    }

}
```

`Adder` class

```
package frigo.dojo;

public class Adder {

    private int result;

    public int result () {
        return result;
    }

    public void add (int value) {
        result += value;
    }

}
```

Mock Objects

Sometimes during testing, the number of collaborating classes required to instantiate a specific class under test is prohibitively expensive. This mainly occurs during Unit Testing, where the tests have very limited scope, one class or unit ideally, must run standalone, must run fast, and should not use slow resources like disks, networks or even databases. However it can arise during other levels of the testing pyramid as well.

It is possible to replace these collaborating classes with test doubles called mock objects. The behavior of these objects then can be predetermined according to the needs of a particular test. Furthermore it is also possible to verify whether the class under test used these collaborating classes in the proper manner.

For example, consider a tabletop game where a player has to throw two six-sided dice to determine the number of steps his figure makes, with special values for (1, 1) and (6, 6) throws. In the computer game implementation of this game, more specifically, in the unit test for the mechanics governing the player's figure, it is possible to mock out the Dice class to test for special boundary values of (1, 1) and (6, 6), and normal values, for example (2, 5), and check whether the figure made the appropriate number of steps, optionally checking whether the dice throw method in the Dice class was called two times.

Spy Objects

Spy objects, or partial mocks are existing objects whose methods have been replaced or extended with mock methods most likely with the use of the proxy pattern. This way it is possible to mock out methods with undesirable behavior or different semantic meaning for the duration of the unit test of the class in question. The usage of spy objects in development can be thought of as an anti-pattern, since a preferred solution would be to extract the method into a separate class, use composition and inject a mock of that class at the start of the test. However when refactoring legacy code, spy objects are very often unavoidable.

For example, unit tests of a texture loader class might mock out the actual loading

of a texture image from the disk to avoid the high cost of disk access and to verify the behavior of the texture loader in cases of various loading failures: When the file can not be found, is corrupt, is not an image, or other I/O error has occured.

Mockito example

This example is a very abstract game example for the purposes of demonstrating mock objects. There is a `Player` who at the start of the game is given an initial health and a regular six-sided `Dice`. When damage occurs, he must roll the dice twice and sum the rolls to determine the damage, receiving no damage at a roll of 2, and double damage, or 24 at a roll of 12.

The `Player` class is tested by injecting a mock of the `Dice` class.

`PlayerTest` unit test

```
package frigo.dojo;

import static org.hamcrest.Matchers.~~is~~;
import static org.junit.Assert.assertThat;
import static org.mockito.Mockito.doReturn;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.verify;

import org.junit.Before;
import org.junit.Test;

public class PlayerTest {

    private static final int INITIAL_HEALTH = 100;

    private Dice dice;
    private Player player;

    @Before
    public void setUp () {
```

```
        dice = mock(Dice.class);
        player = new Player(INITIAL_HEALTH, dice);
    }

    @Test
    public void health_is_initialized_properly () {
        assertThat(player.getHealth(), is(INITIAL_HEALTH));
    }

    @Test
    public void player_rolls_twice_for_damage () {
        player.damage();
        verify(dice, times(2)).roll();
    }

    @Test
    public void rolled_damage_gets_subtracted_from_health () {
        doReturn(3).doReturn(6).when(dice).roll();
        player.damage();
        assertThat(player.getHealth(), is(INITIAL_HEALTH - 3 - 6));
    }

    @Test
    public void player_is_damaged_double_when_rolling_12_as_damage () {
        doReturn(6).doReturn(6).when(dice).roll();
        player.damage();
        assertThat(player.getHealth(), is(INITIAL_HEALTH - 24));
    }

    @Test
    public void player_is_not_damaged_when_rolling_2_as_damage () {
        doReturn(1).doReturn(1).when(dice).roll();
        player.damage();
        assertThat(player.getHealth(), is(INITIAL_HEALTH));
    }

}
```

`Player` class

```
package frigo.dojo;

public class Player {

    private Dice dice;
    private int health;

    public Player (int initialHealth, Dice dice) {
        health = initialHealth;
        this.dice = dice;
    }

    public void damage () {
        int amount = dice.roll() + dice.roll();
        switch( amount ){
            case 2:
                damage(0);
                break;
            case 12:
                damage(24);
                break;
            default:
                damage(amount);
                break;
        }
    }

    public int getHealth () {
        return health;
    }

    private void damage (int amount) {
        health -= amount;
    }
```

```
}
```

`Dice` class

```
package frigo.dojo;

import java.util.Random;

public class Dice {

    public static final int NUMBER_OF_VALUES = 6;

    private final Random random = new Random();

    public int roll () {
        return random.nextInt(NUMBER_OF_VALUES) + 1;
    }

}
```

Clean Code

For the best possible introduction into the world of Clean Code and Test Driven Development, please read Clean Code: A Handbook of Agile Software Craftsmanship by Robert C. Martin.

Clean Code is the art of creating and maintaining elegant, organized, quality code that is easy to read and modify.

Qualities of a source code that is considered clean

- Maintains software development best practices.
- Easy to read, interpret and reason about, clearly expresses the intent of the programmer.
- Eschews complexity in favor of simplicity.
- Does not introduce unnecessary complexity just to be "clever".
- Well tested, with several levels of tests: Unit Tests, Functional Tests, Integration Tests, Acceptance Tests, End-to-End Tests, System Tests.
- Well-formatted, conforms to an agreed upon formatting in case of team projects.
- Maintains proper abstractions of modules and domain specific concepts.
- Names of concepts, classes, methods, objects, parameters, variables are meaningful, well chosen and representative.
- Easy to extend or modify aspects of its behavior in light of changing project goal and direction.
- Strives to solve the project goal rather than getting sidetracked on trivial matters.
- *"Clean code is simple and direct. Clean code reads like well-written prose..."* - Grady Booch, author of Object Oriented Analysis and Design with Applications.

One of the most effective tools to achieve and maintain a state of Clean Code is Test Driven Development.

Test Driven Development

For the best possible introduction into the world of Clean Code and Test Driven Development, please read Clean Code: A Handbook of Agile Software Craftsmanship by Robert C. Martin.

Test Driven Development is a software development process that relies on the repeated application of three short steps, also called *red/green/refactor*:

1) Write a failing unit test, and not more.
2) Write enough production code to make all tests pass, including the previously failing unit test, and not more.
3) *Refactor* the production and test code so it conforms to Clean Code principles, while keeping all unit tests passing.

Keeping to this simple process implies several consequences:

- Development is focused and fast because of the very short feedback cycle. Developers do not start working on slightly related or completely unrelated features, they concentrate their efforts on making the failing unit test pass.
- Unit tests are focused and are inexpensive to run. They do not become slow Integration Tests or End-To-End tests, trying to test a significant portion of the application.
- Code coverage remains high. This is a direct consequence of step 2
- Code quality remains high. This is a direct consequence of step 3.
- The entire suite of unit tests can prove the correctness of the application.

Refactoring

A crucial step of TDD is refactoring. Refactoring is the modification of the software structure without changing its behavior. This is done with the aim to make the source code more conformant to Clean Code principles. This can mean better readibility, reduced complexity, better domain abstraction, better architecture.

Refactoring steps can range from the simplest, like renaming a variable, to the

more complex, like extracting an interface from a class for better abstraction, or moving a subset of fields to a new class to improve cohesion and make the classes less coupled.

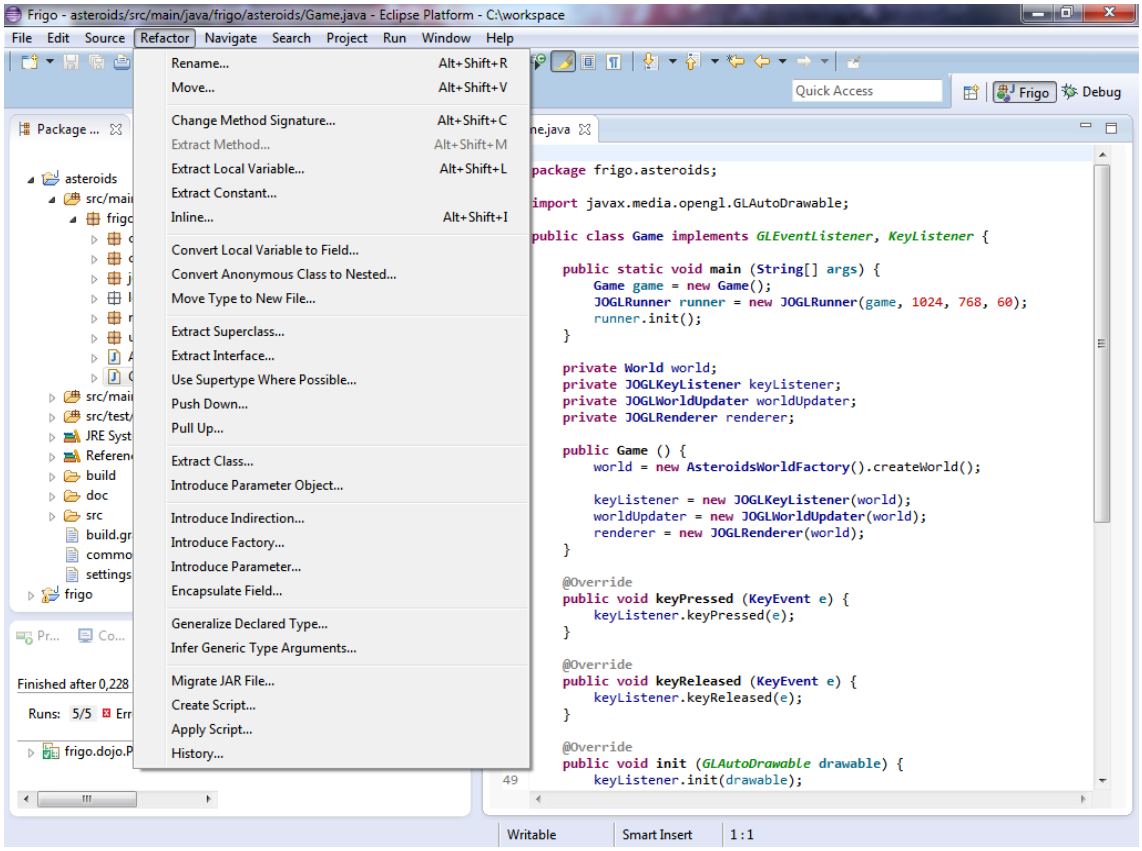
While manual refactoring is possible, it is ill-advised due to its costs and associated possibility of introducing errors. Automatic refactoring tools are preferred.

Common examples of refactor steps from Eclipse

Eclipse is one of the first IDEs offering comprehensive refactor tools and had a huge role in popularizing Refactoring and Test Driven Development.

- Rename: Change the name of a variable, parameter, method, class or package.
- Move: Move a class to another package.
- Change Method Signature: Add, remove or change parameters and their types.
- Extract Method: Extract a section of code and all occurences into a method.
- Extract Local Variable: Extract an expression into a local variable.
- Extract Constant: Extract a constant expression into a static final field.
- Inline (Method): The opposite of Extract Method. Replaces one or all methods calls by the method body.
- Inline (Local Variable): The opposite of Extract Local Variable. Replaces references to the local variable by its initialized value.
- Inline (Constant): The opposite of Extract Constant. Replaces references to the static final field by its value.
- Convert Local Variable to Field: Changes the declaration of a local variable to be that of a field.
- Extract Superclass: Extract a subset of fields and methods of the class into a new superclass and let the class extend it.
- Extract Interface: Create a new interface based on a subset of methods of the class and let the class implement it.
- Push Down: Push down fields or methods into subclasses.

- Pull Up: Pull up common fields or methods into the superclass.
- Extract Class: Extract a subset of fields and methods operating on them into a new class and let the class delegate them to this new class.
- Encapsulate Field: Generate getters and setters for a field.



The Refactor menu of Eclipse.

Revision Control

Revision Control Systems allow developers to keep track of changes done to a document or source code repository in a formalized manner. Its purpose is to supersede ad hoc source control methods like central FTP servers with version named files, code changes broadcasted in emails or other suboptimal solutions.

In its basic form, the source code repository is kept on a central server, where developers can *check out* or *clone* the newest revision to their own private machines in the form of a *working copy*. After developers are done modifying files in their working copy, they can *commit* or *check in* the changed files or changes to the central server, where it is incorporated into a new revision, keeping all older revisions intact.

Should modification conflicts arise, there are several ways of handling them: diff/merge tools, automatic merge, manual merge.

Atomic commits

A revision control system is called atomic if a group of changes is committed all at once, or none is committed at all. That is, two commits can not interleave each other. Subversion, Git, Mercurial are examples of revision control software that are atomic. CVS and ClearCase are not atomic. ClearCase bears special mention as being file-centric rather than repository-centric, that is, it keeps track of versions of file separately rather than having revisions of the entire repository.

Conflicts

There arises the possibility of two or more changes to a file (or directory) conflicting. With text files, changes to different parts of the text file are trivial. However when the same parts of the text file is modified in two different ways, it becomes necessary to either merge them, or prevent the conflict from occuring at all.

Pessimistic and optimistic conflict handling

It is possible to prevent conflicts from occuring at all, with the usage of pessimistic locking. In this case only the one developer who locked a file can make

modifications to it. Others have to wait until he commits his modification or cancels the lock. This approach does not scale well to larger projects, especially when multi-site development is concerned. Examples for this approach include the *reserved checkouts* of ClearCase.

Optimistic concurrency control on the other hand, does not restrict modifications to working copies. As it implies, it tries to solve conflicts as they appear rather than trying to prevent it. Revision Control Systems implementing optimistic concurrency control usually reject commits if they conflict with a previous commit, forcing either client or server side merge of modified files.

Merging

Merging tools usually use the *three-way merge* algorithm. They take the common base version of two modifications and the two modifications themselves, and try to merge the changes so both modifications are applied to the common base.

This is sometimes trivial when the changes do not overlap. However when the changes do overlap, or in case of files where the three-way merge algorithm is not applicable, such as in the case of binary files as opposed to text files, these changes are not applicable.

In the case of conflicts on binary files, it is obvious that manual intervention is required, the developer has to manually incorporate both changes into the new version.

However in the case of text files, it is possible to speed up the process of the manual merge by first using automated merging to exclude trivial changes in the text file, then helping the manual merge phase with a GUI that for every conflicting change offers an option to apply one, both, or neither of the change to that particular conflicting section. This is usually denoted with A, B, C or 1, 2, 3 buttons. The possibility of manually editing the resulting merged file of course remains even with this method.

Labels, tags

It is possible to apply labels or tags to certain revisions of the repository, or revisions of files. For example, it is possible to tag a certain revision as a release revision, for example “1.0.0”, or “1.0rc1”.

Branches

Sometimes it is necessary to split the revision history of the repository in a tree-like manner into branches. This need most often arises when the developers have to maintain multiple releases of the product, for example, fixing a bug in an older release, or fixing a common bug on all branches where there are modifications in the main branch that should not be merged into older branches.

There is also a possibility of using *feature branches*, forking the main branch to implement a specific feature without interference from other developers or development teams. Merging the feature branch back into the main branch however can cause numeruous conflicts.

In the author's personal opinion, branches add more problems than they solve. Every branch is basically a separate product with similar costs to the main branch. They are also subject to Continuous Integration, server costs, Issue Tracking, Documentation, bug fixing, IDE configuration and other activities and responsibilities.

Feature Toggles or Feature Switches are a more promising alternative to Feature Branches, with the additional advantage of dynamic deployments of features and also blind deployments where applicable.

Distributed Revision Control

It is also possible to forego the aforementioned server-client architecture of Revision Control Software, and treat every single working copy, or *clone* as a full-fledged repository. This is especially useful against situations where the central server hosting the repository of the project becomes unavailable for some reason. In this case, developers using Distributed Revision Control can still continue their work, committing their changes to their local repository or to a temporary repository set up on another computer, and only merging the changes when the central repository comes back online.

In essence, Distributed Revision Control removes the central repository as a single point of failure.

Examples of Revision Control Software

- Concurrent Versions System

- Subversion
- Git
- Mercurial
- ClearCase

Build Automation Software

Build Automation Software are aimed at automating various steps of repetitive Software Development activities. These include but are not limited to:

- Compilation of source files into binary code
- Packaging of binary code into software packages, for example into .jar files
- Running various levels of tests: unit tests, functional tests, end-to-end tests, etc
- Publishing the resulting libraries and software to software repositories, for example to a Maven repository
- Deployment of resulting software to production systems, for example to a demo server

Some features and responsibilities of Build Automation Software overlap with Continuous Integration.

Examples of Build Automation Software

Make

A build automation tool with a long history. It remains widely used in the Unix/Linux world. It uses its custom Makefiles to describe build steps.

Unfortunately it is somewhat deprecated by newer tools and does not provide adequate support for Java projects.

Apache Ant

A Java based multiplatform tool providing build process automation based on scripts in XML format.

In the author's opinion it places too few restrictions on the build process and as such build scripts of complex software tend to get inconsistent and unmaintainable.

It has no concept of modules and it does not have native dependency management which severely limits its use. Its state is not persistent, successive Ant calls can result in

unecessary recompilations and other redunant activities.

Apache Maven

Also a Java based multiplatform tool providing build process automation based on scripts in XML format.

Contrary to Ant, it severely restricts the developer's choices regarding build automation. It enforces the use of modules and its own module management. It has a steep learning curve, often difficult for a new project. It is not customization friendly.

In the author's opinion its complexity and unreliability, colloquially often called “Maven Hell”, severely limits its use in both small and large projects and as such the author suggests avoiding its usage.

Gradle

A Groovy based multiplatform tool providing build process automation based on custom Groovy scripts. Often perceived as a middle path between Ant and Maven.

Its main disadvantage is its uncommon Groovy syntax that is not trivial to learn for developers new to Gradle. However, given familiarity with the syntax and the reasoning behind Gradle constructrs, it can become quite easy to write or maintain scripts. It is also compact and expressive unlike its XML based cousins.

The author chose Gradle as the Build Automation Tool best suited for this project, based on his experience with Ant, Maven and Gradle.

Continuous Integration

Continuous Integration is the practice of using a central server to merge and check, or *integrate* all developer contributions to the software project as often as possible.

The goal of Continuous Integration is to prevent *Integration Hell* or *Merge Hell* that is characteristic of periodic integration.

Integration Hell is the conflict pileup of developer contributions to the project. It inherently arises during parallel development by multiple members or teams working on the same codebase.

By requiring developers to integrate as often as possible, these conflicts are detected and fixed as soon as they arise, rather than waiting for days or weeks for conflicts to stack up into an unmanagable pile.

Continuous Integration systems use Revision Control Systems to get the newest revision of the project, use Build Automation Software to compile, test and build the application, and use Artifact Repository Systems to store the resulting artifacts, among other activities.

Daily builds

Several build and unit test runs of the project every day is a basic requirement of Continuous Integration. Preferably every time someone commits a modification. It is imperative that these basic builds and unit tests are fixed as soon as they fail. Obviously, these builds have to be fast to have fast feedback on compilation errors and other failures.

Nightly regression

However, given a large enough application, it is impossible to run every single test in the fast daily builds. In this case it is necessary to introduce nightly regression jobs.

These jobs usually run tests that are of slower nature than Unit Tests: Functional Tests, Integration Tests, End-To-End Tests and System Tests.

In an ideal situation, they set up a clone of the production environment to (virtual) servers and clients, install the latest deliverables produced by the daily builds, and run the aforementioned tests on them.

Other Activities

Continuous Integration can do other tasks that are not inherently part of the Deployment Pipeline. Automated Software Quality Control is one such task that is not inherently needed as a requirement for deployment, but is obviously preferred.

Jenkins

Jenkins is a Java based Continuous Integration tool with a web based Graphical User Interface running on an Apache Tomcat web server.

Jenkins-Hudson split

Jenkins was created as a fork or continuation of the Hudson project after Sun Microsystems was bought by Oracle. Reasons for this split were problems involving the java.net infrastructure where Hudson was hosted at the time, and questions regarding the control of the Hudson trademark by Oracle.

Jobs

Jenkins can be used to create, configure, run and schedule automated jobs (tasks or projects) of any nature, not just build automation for software development. For example, the author has a job responsible for hard drive backup onto a secondary one.

Jobs can be configured via a clean, simple and intuitive modular interface, or manually via XML configuration files, though this is not recommended.

Jobs can have upstream and downstream jobs as dependencies. Downstream jobs are automatically executed when one of their upstream jobs are finished. Jobs can also call each other directly. This way complex or common jobs can be separated into multiple ones. For example, there could be a job that builds an application, and there could be a downstream job that runs functional tests on the resulting distribution.

Jobs can be grouped into tabbed Views, allowing a single instance of Jenkins to

host several projects and all of their related jobs without confusion.

Builds

Jenkins keeps the build history of all jobs until the configured amount of time or number of builds, including all published artifacts and the entire log of the build process. Builds can be tagged to keep forever and labeled with specific release version or other messages.

Builds successfully finishing their execution without any error or failure are *stable* and have *success* as their status. Those encountering an error or failure during their execution are *broken* and have *failure* as their status. Failing unit tests, depending on settings, can cause the build to become either broken, or *unstable*. Jobs can also be disabled so they can not be executed.

Slaves

Jenkins have the option to execute jobs on *slave* nodes rather than on the server running Jenkins. This essentially offloads the vast majority of the work from the server and also enables load balancing. Slaves are also useful for testing the application on a different platform.

Plugins

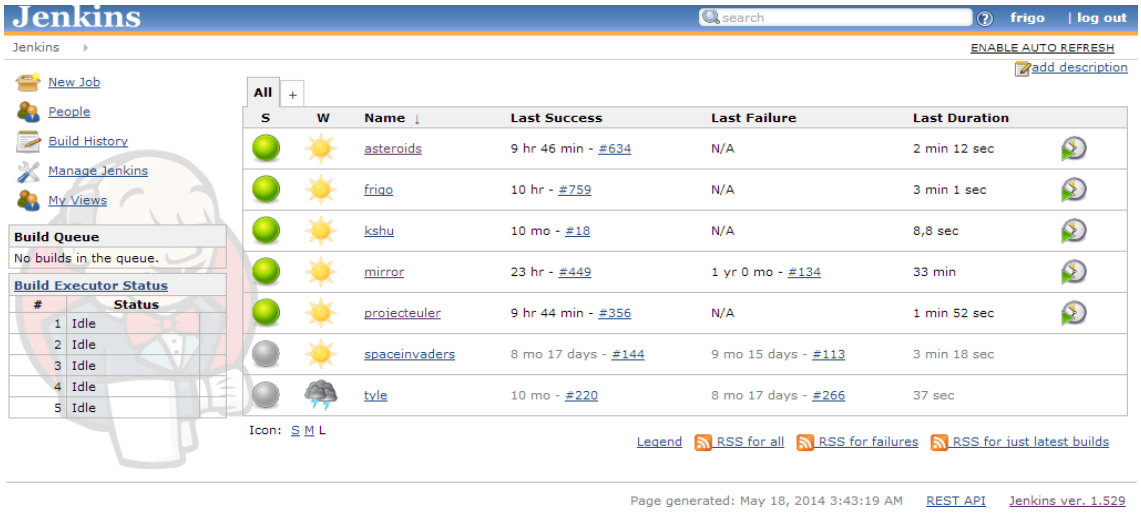
Jenkins is highly modular, it supports many features and technologies either natively or via the support of plugins. A few examples of Jenkins plugins:

- Ant Plugin: Enables Jenkins to run Ant build scripts as part of a job.
- CVS Plugin: Adds support for Concurrent Versions System as a source code management option.
- Email-ext plugin: Enables Jenkins to send customizable emails on job success, failure or regression.
- External Monitor Job Type: Adds a custom type of job that enables Jenkins to monitor the result of externally executed jobs.
- Green Balls: The default color scheme of Jenkins contains red and blue spheres

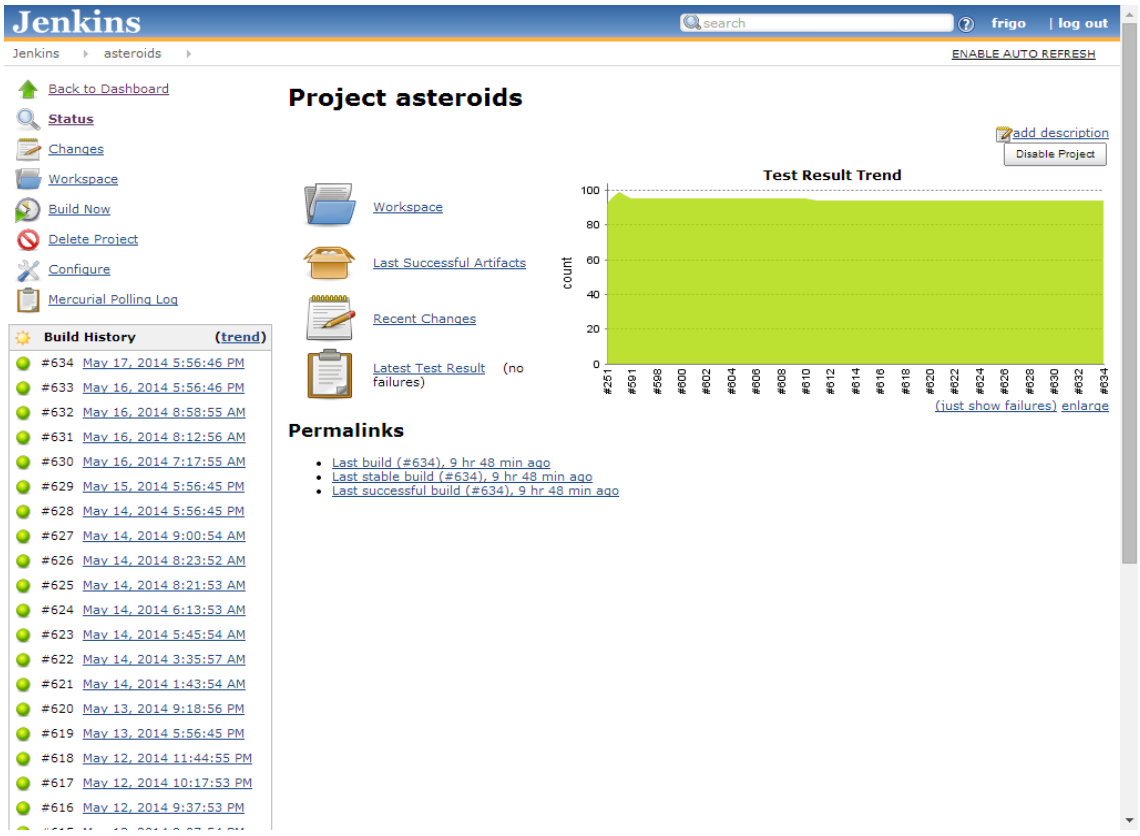
to represent failing and succeeding builds respectively in support for color blind people. This plugin changes the blue spheres into green ones.

- JaCoCo plugin: Adds support for code coverage reports generated by the code coverage software JaCoCo.
- Javadoc plugin: Makes it possible to publish Javadoc documentation on Jenkins.
- Jenkins Gradle plugin: Enables Jenkins to run Gradle scripts as part of a job.
- Jenkins Sonar plugin: Makes it possible to run a SonarQube code quality analysis as part of a job.
- LDAP plugin: Adds support for LDAP based authentication.
- Mailer: Enables Jenkins to send emails on job success, failure or regression.
- Maven Project plugin: Adds support for the build automation tool Maven.
- Mercurial plugin: Adds support for Mercurial as a source code management option.
- PAM Authentication plugin: Adds support for Unix Pluggable Authentication Module based authentication.
- Release plugin: Adds support for release type builds and custom pre- and post-build actions that are executed when a release build is manually triggered.
- SSH Slaves plugin: Allows Jenkins to offload jobs onto SSH-enabled worker nodes to balance load.
- Token Macro plugin: Allows user defined variables in job configurations.
- Translation Assistance plugin: Allows users to contribute translations of messages found throughout the graphical user interface of Jenkins.

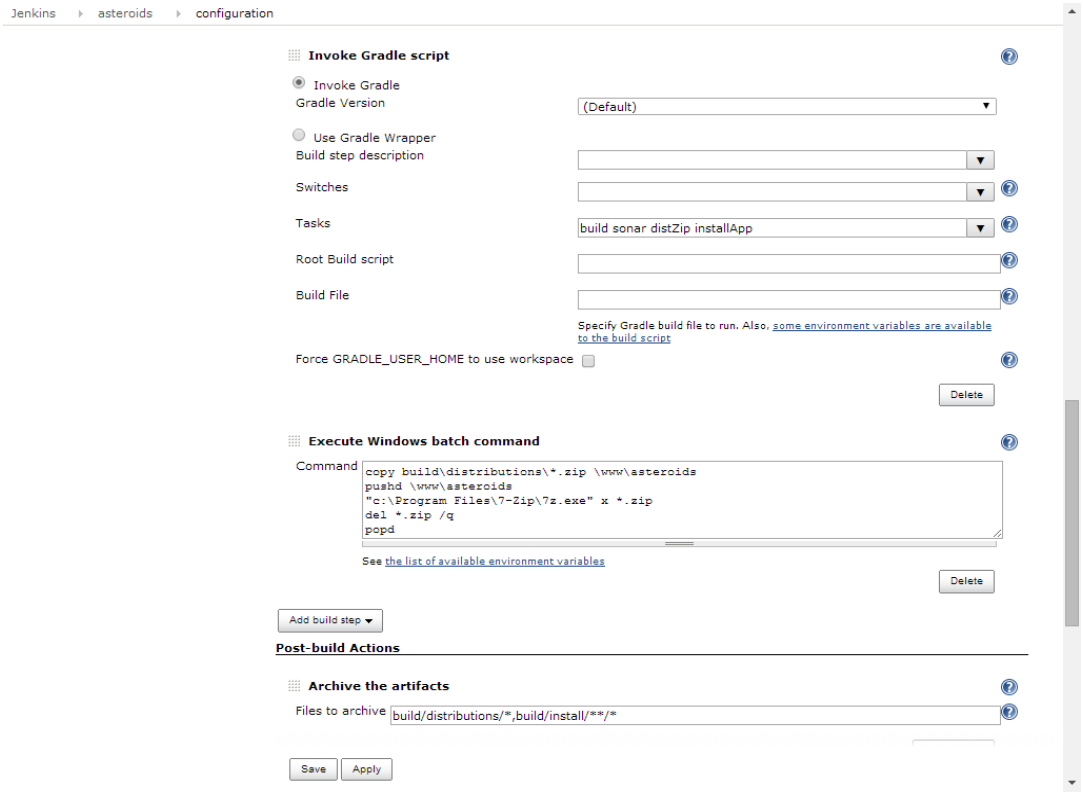
Screenshots



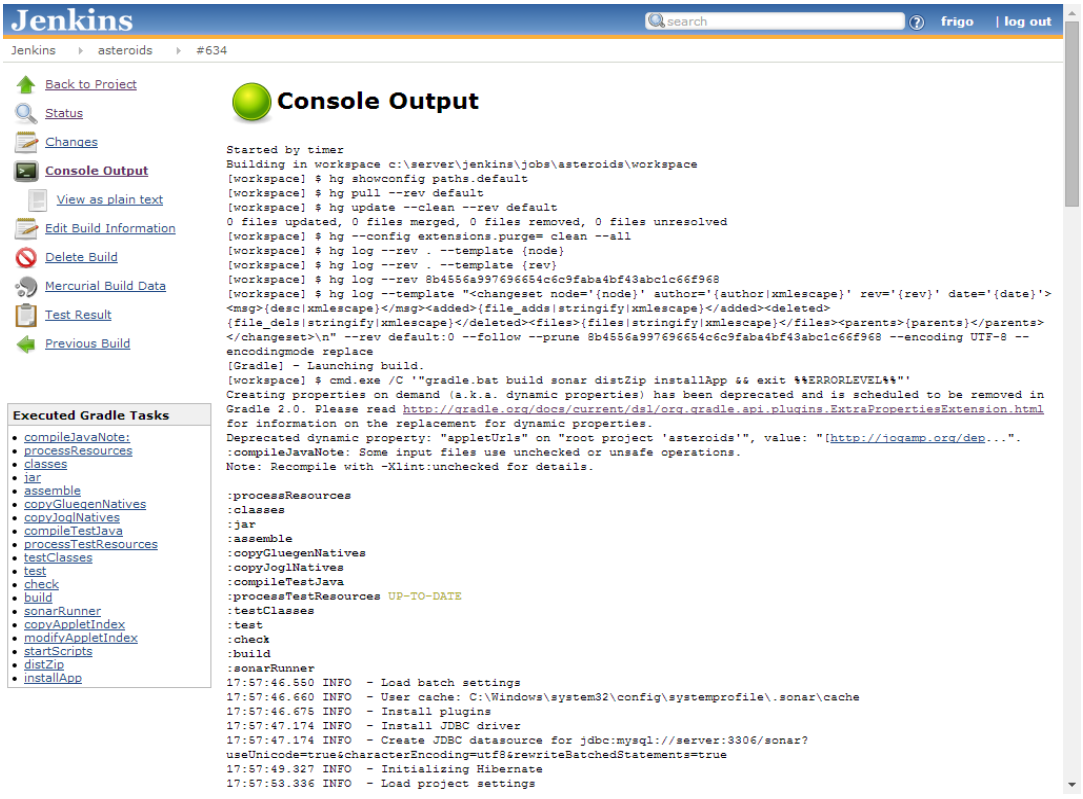
Main page of a Jenkins instance



Project page of Asteroids



Configuration page for project Asteroids



Log of a successful run

Software Quality Control

Software Quality Control is the management and enforcement of the software quality of a project, as measured by appropriate processes, standards and metrics.

Depending on the particular definition, it can include Quality Assurance, Software Testing, Software Reviews and other Software Development processes, or just Automated Software Quality Control. In this chapter it is assumed Software Quality Control is only concerned with the technical details of a software project, and not the various processes and feedback loops encountered during software development.

Automated Software Quality Control

Automated Software Quality Control, or Continuous Inspection, is a type of Software Quality Control that emphasizes the use of automated code quality metrics to determine the overall quality of the source code of an application.

It can be thought as the automated version of Software Quality Control, in a similar way Automated Software Testing is an automated version of Software Testing.

Whereas Software Testing is primarily concerned with the correct meaning, or semantics of the software product, Automated Software Quality Control is primarily concerned with the use of correct technical details, solutions and patterns in a project.

Automated Software Quality Control is not a replacement for Software Testing, but rather an adjunct to it, since they are concerned with different aspects of the project.

Technical Debt

Incorrect technical details, improper solutions, anti-patterns and bad quality in general all contribute to the *technical debt* of a project that over time makes it more expensive or even impossible to develop or to maintain the project.

Static and Dynamic Code Analysis

Software Quality Control software can be static or dynamic. *Static Code Analysis* tools only consider the source code, as a static material to work on, while dynamic code analysis tools can also derive information about the execution of the tests or the

application itself. For example, a tool checking for questionable syntax in a source code is static, while a test coverage analysis tool is dynamic.

Examples of Automatic Software Quality Control software

- Checkstyle: A static code analysis tool checking Java source codes for coding rules compliance. Its rules are described by *modules*.
- PMD: Also a static code analysis tool checking Java source codes for possible problems. It contains *Rulesets* with rules based on either XPath or Java classes.
- FindBugs: A static code analysis tool that works on Java bytecode rather than source code. Its rulesets can also be extended.
- SonarQube: A Java based Graphical User Interface and server that encompasses the previous three tools with additional features.

Software Metric

Software Metrics are a quantitative measure of certain qualities or properties of a given piece of source code.

Examples of software metrics

- Code coverage: Shows the degree to which the source code is exercised or covered by tests. A high coverage is desirable, since it implies that the project is well tested and is robust against regressions.
- Cohesion: Describes the degree to which the elements of a class or package belong together. A high degree of cohesion is desirable, since it implies semantical correctness and proper abstraction levels.
- Comment density: Measures the comments to source code ratio. A good comment density is neither too high nor too low.
- Coupling: Measures the dependencies between classes or packages. A low coupling is desirable, since it implies better separation or abstraction of modules, and thus it is easier to modify or extend their behavior without affecting other modules.

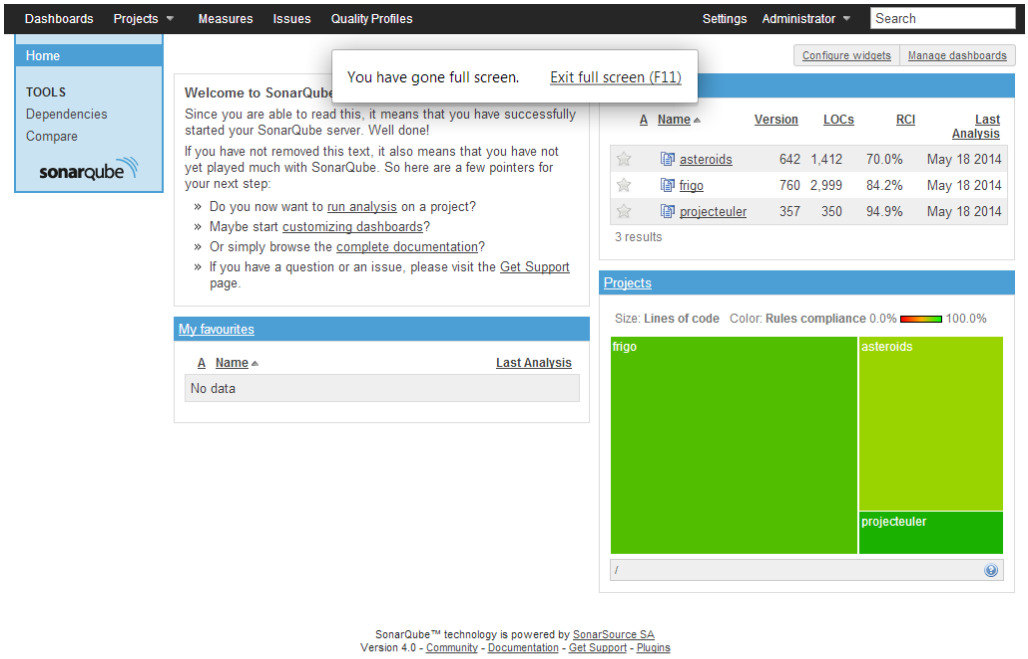
- Cyclomatic complexity: The number of execution paths a method can take during its execution. A low cyclomatic complexity is desirable, since it implies methods are cleaner, simple to reason about, have less conditionals and thus need less tests to ensure the correctness of all execution paths.
- Number of classes and interfaces: The number of classes and interfaces found in the project. A value that is too high can make the project unmaintainable, a value that is too low implies that the abstraction level of the project is improper.
- Number of lines of code: Also called Source Lines Of Code. It is a rough estimate of the complexity of a project or effort required to build it. Made obsolete by other complexity metrics. Frequently misunderstood and misused by both management and developers, highly prone to manipulation, and thus its use is highly discouraged.
- Program execution time: Performance of the application. High performance software is desirable, but not to the exclusion of other aspects of the software project, like testability, proper abstraction levels or maintainability.
- Program load time: The time required to load data, libraries and resources required to put the application in a working state.
- Program size: Size of the resulting application binary. Similarly to program execution time, it should not be improved to the exclusion of other aspects of the software project. It is much less of a concern with modern computers, with the possible exceptions of embedded software and restricted runtime environments.

SonarQube

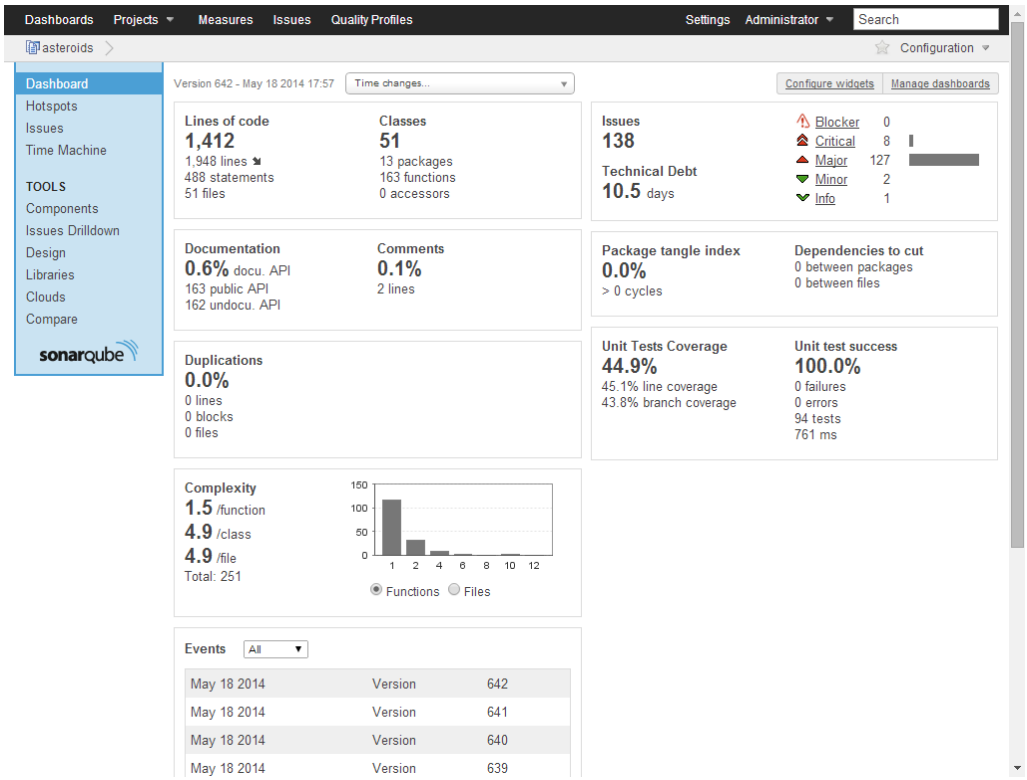
SonarQube, formerly called Sonar, is a Continuous Inspection software written in Java. It has a web based Graphical User Interface for project, issue and profile or ruleset management. It uses a server-client architecture where the clients, called *Sonar runners*, send the result of their analysis to a central server and its database.

It incorporates the results of Checkstyle, PMD, FindBugs, JUnit results and code coverage analysis tools.

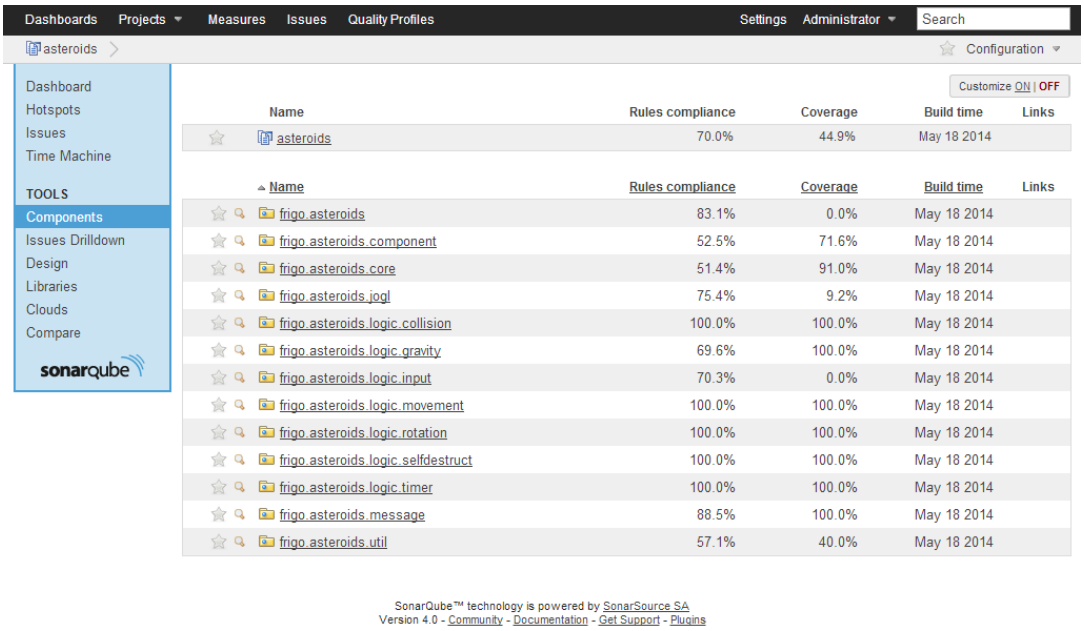
Screenshots



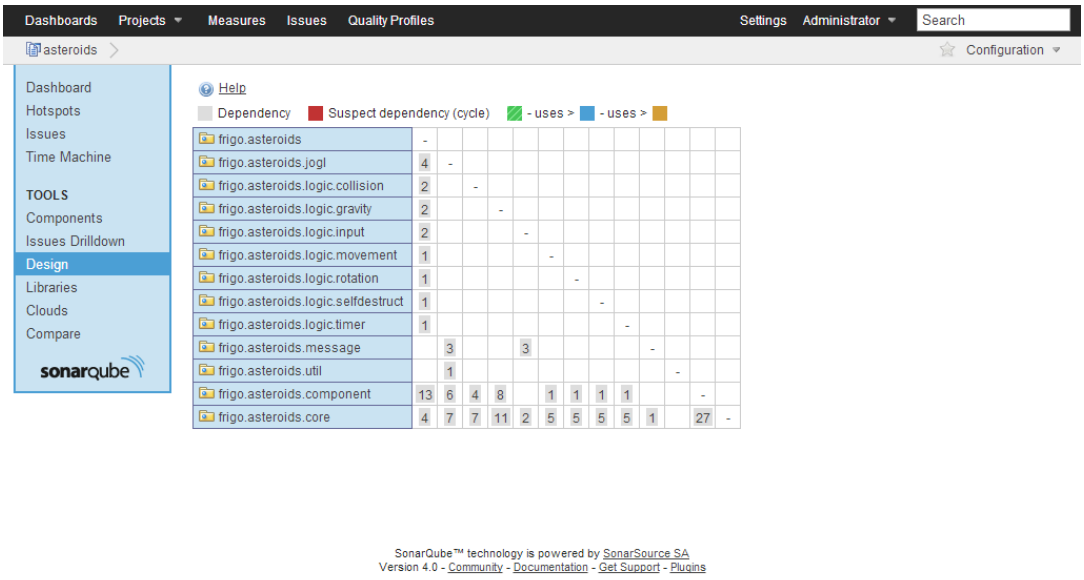
Main page of Sonar.



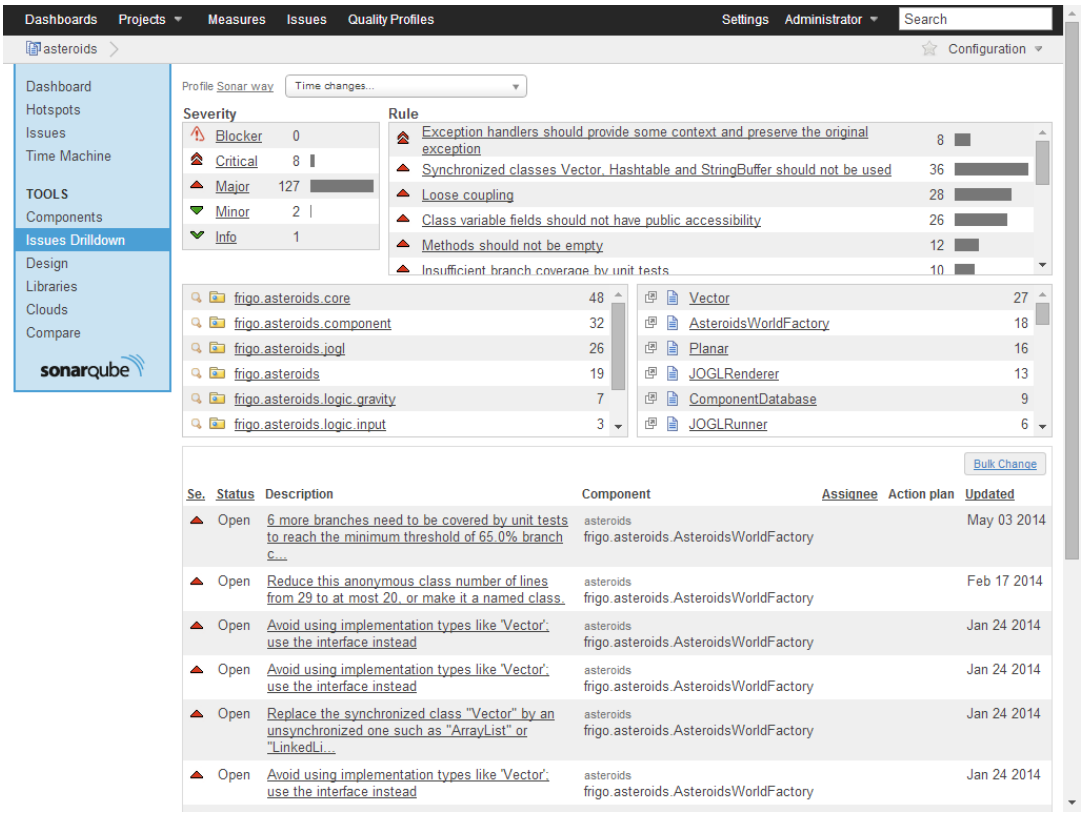
Software quality metrics of the Asteroids project.



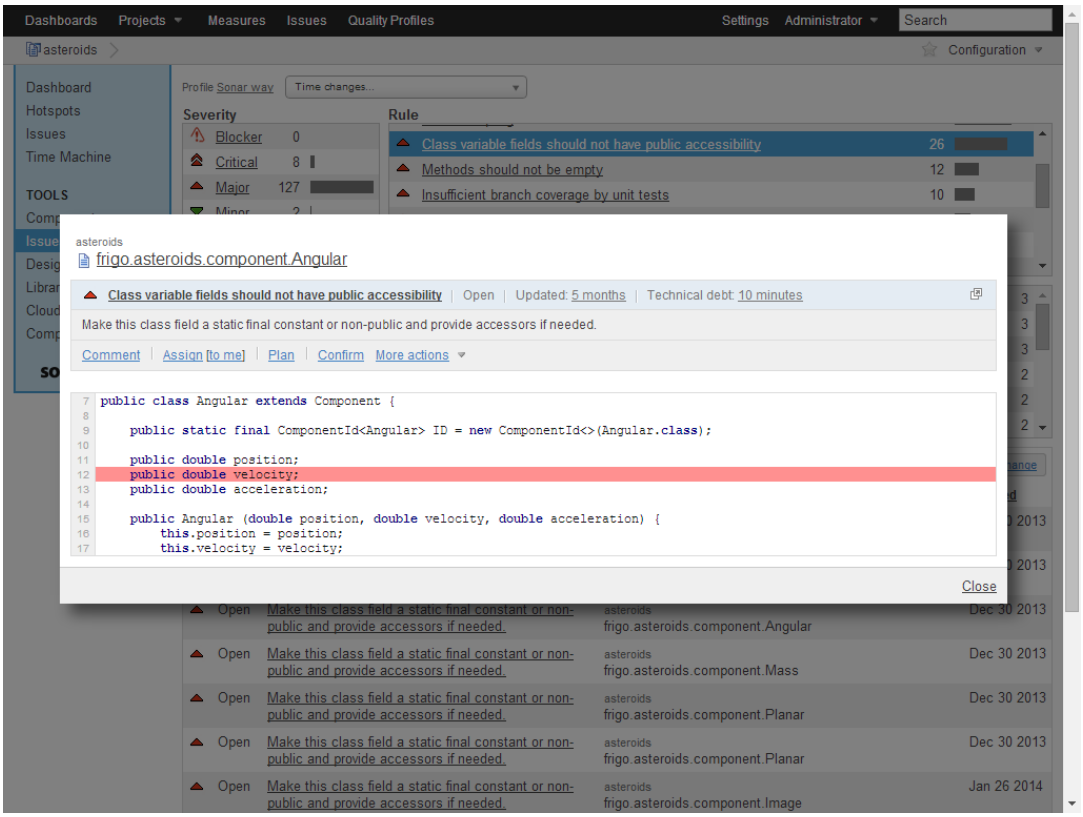
List of packages in the Asteroids project, their rules compliance and coverage.



Package dependency matrix used to determine the package tangle index.



List of issues in the Asteroids project.



An example issue, or rule violation.

Issue Tracking System

An Issue Tracking System, or Bug Tracking System, is a software system that provides a semi-formalized way for managing and maintaining a list of *issues*, or *tickets* associated to a project or several projects that needs to be solved during the lifetime of the project or the release.

Examples of Issue Tracking Systems include:

- BugZilla
- JIRA
- Redmine
- Trac

The Asteroids project uses the minimalistic JIRA version that Bitbucket provides for hosted projects.

Issue

An issue can be any of the following:

- Feature request or enhancement
- Bug
- Missing documentation
- Missing test case
- Refactor task
- Proposal
- Task

Priority

Issues normally have priorities assigned to them to express their urgency, based on severity, scope of effect, and customer impact. For example, a bug that makes it

impossible to deploy a software at a customer is a catastrophic failure, affects the entire application, directly impacts the customer, and thus it is very urgent to fix and is given a Blocker rating.

Usual levels of priorities:

- Trivial: Cosmetic bugs that are trivial to fix and have basically no impact. For example, a typo on the user interface of the software.
- Minor: Bugs with very limited scope that do not normally hinder the usage of the application.
- Major: Bugs that affect the normal working of the application or modules but have workarounds that keeps the application functional.
- Critical: Critical bugs prevent the application or modules from working properly and there are no workarounds for them.
- Blocker: Blocker bugs prevent the entire system from working at all. They need to be fixed immediately, development simply can not continue until blocker issues are resolved.

Depending on the Issue Tracking System used, there may be other priority levels. Either less for the sake of simplicity and practicality, or more to express special situations.

Assignee

An issue usually have a developer or maintainer assigned to it by the submitter or reporter. It is the responsibility of the assignee to see the issue to completion to the satisfaction of the submitter or previous assignees. By default the project manager is assigned new issues who can then reassign them. However this is not necessary, anyone related to the project can be assigned issues, for example members of the customer support team are often targets of new issues.

Status

Issues can have a status assigned depending the progress already done.

Examples of status values:

- New: Automatically assigned to new issues
- Open: The issue is accepted but has not been resolved yet.
- On Hold: The issue is put on hold. Possibly due to being out of scope for the release.
- Resolved: The issue has been resolved and awaiting confirmation by the reporter, other assignees, or the program manager.
- Duplicate: The issue has been reported before, the new one is basically a duplicate of the old one.
- Invalid: The issue is invalid. Possibly due to being too vague, impossible to understand by the assignee, or not related to the project.
- Won't fix: The issue is simply not worth the time
- Closed

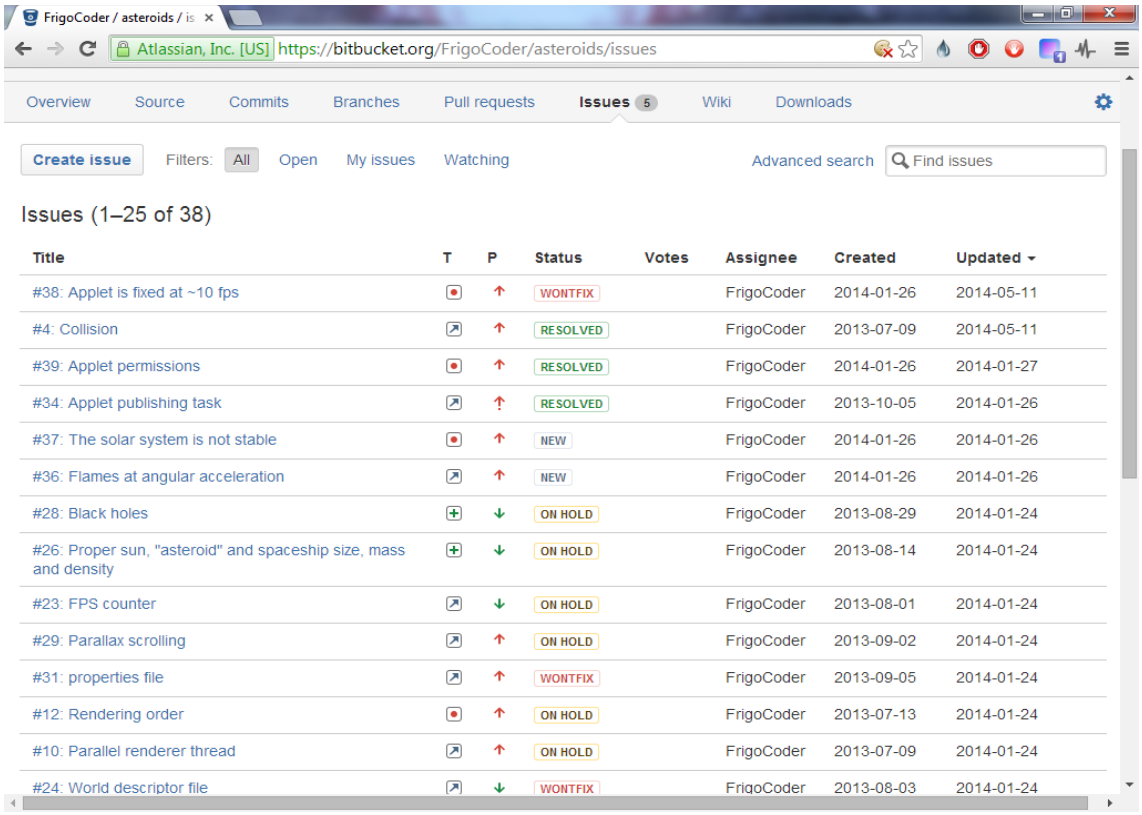
JIRA

Atlassian JIRA is a proprietary Issue Tracking System created by Atlassian. It is written in Java and uses a web based Graphical User Interface. It is available free for use by non-profit organizations, charities and open source projects.

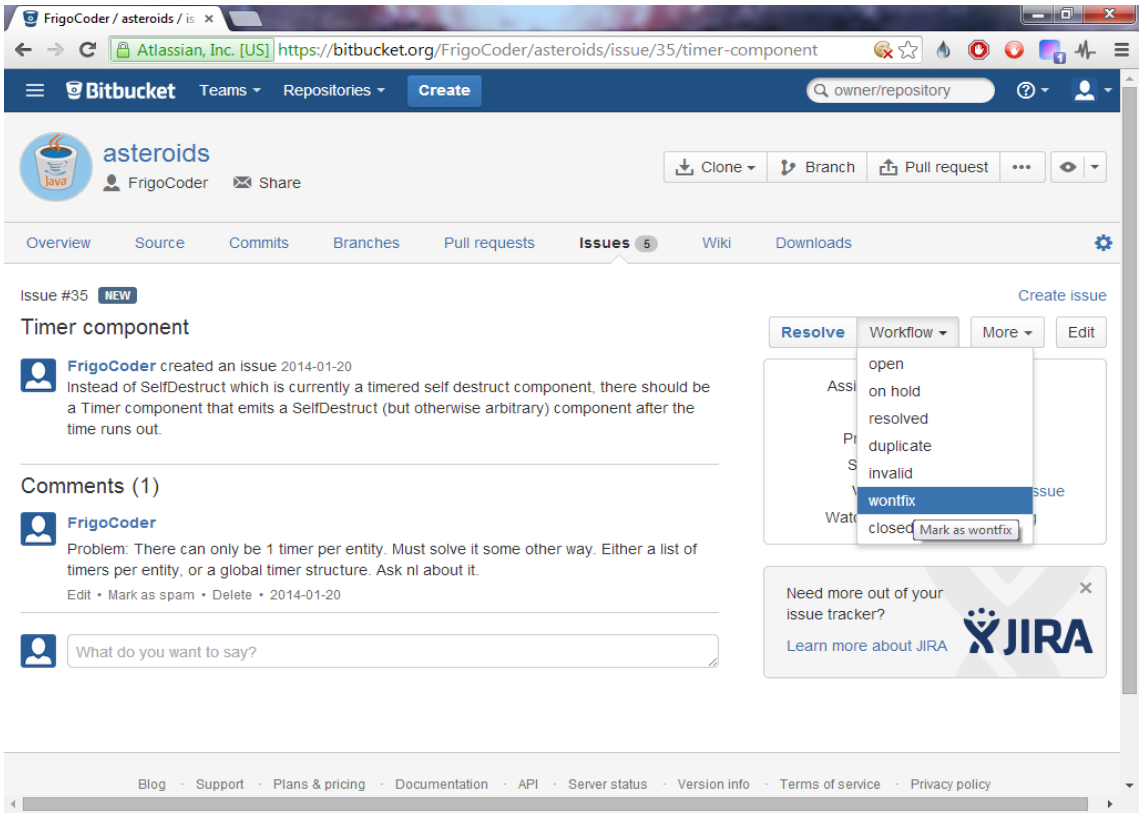
It supports several Revision Control Systems, including Subversion, CVS, Git, ClearCase and Mercurial among others.

It has plugins integrating its project development workflow into Integrated Development Environments, including plugins for Eclipse and IntelliJ IDEA.

Bitbucket has a minimalistic installation of JIRA that is used by Asteroids.



JIRA for Asteroids showing a list of issues



Closing an issue with the Won't fix status.

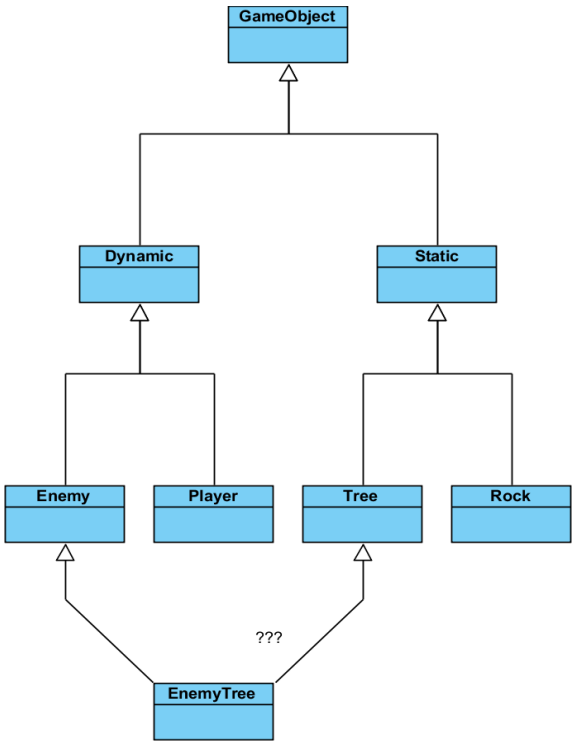
Entity Component Systems

The Limitations of Object Oriented Programming

Object Oriented Programming, while often the best choice for a wide variety of programming tasks, is particularly unsuited for a set of problems where the decomposition required by the problem differs from the hierarchical decomposition offered by Object Oriented Programming. One such problem is encountered in game development: Representation of game objects.

Inheritance and deep class hierarchies

Object Oriented Programming implies inheritance and deep class hierarchies. When applied to game objects, this translates to a class hierarchy that is basically arbitrary and does not correctly decompose the concerns (or aspects) shared by sets of game objects. Concerns of game objects are not decomposable in a hierarchical manner.



This example shows one such scenario. There are two aspects of game objects, one is whether they are static or dynamic, and one is whether they are enemies or not, that can not be accurately represented in a hierarchical single inheritance model.

Enforcing such an arbitrary class hierarchy invariably results in inconsistencies and subtle bugs.

Static class hierarchies

Static class hierarchies are also resistant to change. When a new game concept is introduced that does not fit into the existing class hierarchy, in the worst case the entire class hierarchy has to be reworked, implying the modification of virtually all game object classes in the project. This results in increasing difficulty of incorporating ideas into the game as the project goes larger.

Static object types

Static class hierarchies have another limitation: Objects can only have one type or class that can not be changed during their lifetime. Keeping to the example above, it would be very difficult for an EnemyTree to pull his roots out of the ground, move to another place and replant himself, in other words, to switch between being Static and Dynamic. A common workaround for this problem is to create a new game object instance every time such a switch occurs. However this approach does not preserve the identity of the game object, can lead to inconsistencies and subtle bugs, and given enough such switches, it is prohibitively expensive.

A real life example for such a bug can be found in Starcraft: A Drone, a worker unit, can be transformed into an Extractor, a building used to collect resources, which then can be canceled before finishing, giving a drone back to the player. However, this unit is a new one, and it does not preserve the health of the old one, furthermore, enemy Artificial Intelligence loses track of the old drone.

Encapsulation and polymorphism

While from a maintenance point of view, encapsulation, or coupling of data and methods operating on said data might look good at first, in reality it often interferes with the flexibility of maintenance. Adding new data, modifying existing ones, or modifying application logic can often mean lengthy and error-prone upgrade processes. Classes are not inherently prepared for such modifications. Even in a database where data and application logic is inherently separated, these updates are not trivial and without cost.

Object-Relational Impedance Mismatch

Objects and Classes are also ill-fitting for databases, whether relational or semi-structured. This mainly stems from several reasons:

- Classes are inherently encapsulated, while databases either only store data, or store data separately from logic.
- Classes strive to provide *interfaces* through which they can be manipulated, databases provide *raw data*. Classes are behavior oriented, while databases are data oriented.
- Identity in Object Oriented Programming is implicit, while databases do not differentiate between two records or rows with the same information, requiring the introduction of identifiers.
- Object oriented concepts like classes, inheritance, polymorphism, aggregation, composition and others do not exist in the relational database world, and can be achieved only with workarounds.
- Classes and database schemas do not have a one to one mapping.

Object Relational Mapping

There have been several unsuccessful attempts at providing *Object Relational Mapping* solutions to the impedance mismatch between Object Oriented Programming and Relational Databases. However, most of them suffer from some logical inconsistency or other problems and are not recommended for serious production quality systems.

For example, a trivial workaround is to persist objects into the database as byte streams. This degrades the database to a persistency layer, and abandons several advantages of databases like transactions or atomicity of operations.

Alternative database models

A more promising approach is to abandon Relational Databases, and use any of the following database models:

- Object databases: Specifically created for storing objects in an Object Oriented like manner.
- Graph databases: They represent objects and their associations as a graph.
- Document databases: They store objects as dynamic documents, or collection of fields. XML and JSON based databases fulfill this model.
- Key-Value stores: They store only keys and associated values.

However, these models still do not provide a fully consistent, transparent, end-to-end mapping between objects and raw data in databases.

As per the reasons described above, let's introduce an alternative to Object Oriented Programming.

Basics of Entity Component Systems

Entity Component Systems approach the problem of modeling concepts and processes with the use of *entities*, dynamic aggregates or compositions of *components*, and *systems* containing the logic rather than coupled to the data.

This is in contrast to several Object Oriented Programming principles, like classes, objects as fixed instances of classes, encapsulation, polymorphism, inheritance and static class hierarchies.

Entity

An entity is an identity with a set of associated component instances.

For example, an asteroid in the game is a randomly assigned integer identity with `Angular`, `Attractable`, `Health`, `Image`, `Mass`, `Planar` and `Size` component instances.

The most basic implementation of an Entity would be a Map<ComponentId, Component> structure with ***`has`***, ***`get`***, ***`add`*** or ***`set`*** and ***`remove`*** methods.

In Asteroids, Entities do not directly contain Components, they rather refer to a ComponentDatabase instance that stores similar Components contiguously in an open-addressed hash map. This improves both memory usage due to less overhead of smaller sets and performance due to the contiguous processing of similar Components.

Components

Components are data with implied meaning, but without any application or game specific logic.

For example, a **Health** component stores the number of health or hit points of an entity, but contains no logic regarding how damage might happen to the entity.

Obviously basic methods operating on the component as a *data type* can still be stored in the class of the component.

For example, a `Vector` component implying the positon of an entity in a three dimensional space might still contain methods for rotation, translation, addition and other arithmetic functions.

Systems

Systems contain application or game specific logic.

Systems operate on all entities with a given composition of components. They have to be registered into one or more system manager that calls them with the applicable list of entities. This manager can be tied either to a game loop, on-change events of entities, or other events.

In Asteroids, Systems are registered into the game World, where they are called from the game loop in a deterministic order.

Communication between Systems should happen via Message Passing rather than direct Method Invocation or Callbacks.

Advantages of Entity Component Systems

Decomposition of concepts

Game concepts have a clear mapping to Components. There is no need to introduce decomposition of game entities to arbitrary classes and an arbitrary class hierarchy based on some poorly defined semantic criteria.

For example, the problem posed by EnemyTree that was described earlier, can be easily solved by introducing Enemy, Static and Tree components that do not derive from

any superclass nor do they contain any game specific logic.

No static class hierarchies

We do not need class hierarchies either to represent game objects. Any maintenance work will only have to be done on the Component or System in question, and not an entire class hierarchy composed of rigid, static classes.

Dynamic components

Entities have a *dynamic* set of components rather than a single fixed type or class, and these components can be added or removed during the game, leaving open the possibilities for complex game mechanisms.

Keeping to our EnemyTree example, it is possible and actually very easy to remove the Static component and add a Dynamic one when the EnemyTree pulls out its roots from the ground and transforms into a dynamic, moving enemy.

Upgrade friendly

Introduction, removal or modification of a Component is much easier than in the case of Object Oriented Programming. It can be even done dynamically, while the application is still running. Components are (semi-)independent and can be manipulated without disturbing other Components or the application or game logic.

Upgrade of the application or game logic is likewise very trivial. Any change to the game logic does not need to touch Components.

Database friendly

Entity identifiers and Components are naturally a better fit for most database models. They can be mapped the most directly to Document databases, but there are also trivial ways to store them in Key-Value stores or Relational databases as well.

Disadvantages of Entity Component Systems

Communication Between Systems

The method of communication between Systems is not a trivial problem. They can be done with information passed through Components, Entity Groups, World-wide Messages, Special Entities, Callbacks or some other manner. It is not clear which way is the best suited for game development, let alone for arbitrary application development.

If the Systems have a predetermined, strict order, like in the case of game loops, callbacks are very much inadvisable, since they can make the behavior unpredictable, especially if we consider custom Systems added later to the game loop.

Component based messaging is preferred when messages are limited in scope to a single Entity, or separately to several Entities. However when messages are more far-reaching, they might not be the best choice.

Entity Group based messaging requires the definition and handling of Entity groups in the engine. For example, commands given to a group of soldiers require programmatical management of said group of soldiers.

World-wide Messages are debatable whether necessary or not. The author chose this way to implement JOGL keyboard input actions, because of the difficulty and possible anti-pattern nature of passing Entities to external wrapper systems or selecting a special entity for handling external messages.

Special Entities are Entities that are selected in some manner as routers of Messages. The author does not yet have experience with them.

Selecting Entities

There is also the problem of selecting the appropriate Entities for processing by a System using the fastest way possible. There are basically two ways to do this.

One way to do this is to iterate over all Entities for all Systems and decide whether the Entity is applicable to the particular System.

It is also possible to maintain a list of Components applicable to each System, and updating them when a Component is added or removed from an Entity.

Custom logic

It can be cumbersome to add customized logic to Entity Component Systems, for example as part of a customization, mod, or just normal game logic of a game.

For example, Asteroids still uses `InputAction` callbacks to handle various input events and `CollisionAction` callbacks to handle collisions, which do not really fit into the Message based architecture of Systems.

The author is unaware of a completely satisfactory solution to this problem.

Other Tools And Software Libraries used in the project

Apache Commons

A collection of Java utility and syntax sugar libraries.

Awaitility

A library used for testing parallel code.

Google Guava

Java utility and syntax sugar libraries.

Gluegen

Provides Java interface for C/C++ libraries. Used for JOGL.

Hamcrest

Matchers for JUnit and Awaitility.

HPPC

High Performance Primitive Collections library.

JOGL

OpenGL wrapper for Java.

Log4j

Logging library.

SLF4J

Interface for logging libraries. Used by Log4j.

Structure of Asteroids

Directory structure

.gradle/	Temporary storage for use by Gradle
.hg/	Temporary storage and settings for use by Mercurial
.settings/	Eclipse settings
bin/	Binary files as compiled by Eclipse
build/	Binary files and temporary storage for use by Gradle
classes/	Class files
dependency-cache/	Local dependency cache differing from main cache
libs/	Compiled jar files of the projcet
resources/	Non-Java resources
scripts/	Scripts to start the game under Windows and Linux
tmp/	Temporary storage for use by Gradle
doc/	Documentation and resources used to construct the thesis
src/	Source files
applet	HTML template for Java Applet version of game
doc	Resources used to construct the thesis
main	Sources for the *main* module
java	Java sources for the *main* module
resources	Image and other resources for the *main* module
test	Sources for the *test* (unit test) module
java	Java sources for the *test* (unit test) module
.classpath	Eclipse classpath information, generated by Gradle
.hgignore	List of files to ignore for Mercurial
.project	Eclipse project, generated by Gradle
build.gradle	Gradle build script for the project
common.gradle	Gradle build script containing cross-project utilities
settings.gradle	Gradle build script required to set name of root project

Package structure

frigo	Common package name used by the author
asteroids	Package name created specifically for Asteroids
component	Component classes used by entities
core	The core ECS engine and associated data types
jogl	JOGL specific input- and rendering systems
logic	ECS Systems
collision	Collision system and associated interfaces
gravity	Gravity system and associated classes
input	Input system and associated interfaces
movement	Movement system
rotation	Rotation system
selfdestruct	Self-destruct system used by explosions
timer	Timer system

message	Messages used both externally and internally of core
util	Java syntax sugar utilities

Logical structure

Asteroids is split into these logical modules:

The main application class Game that wires the Asteroids specific game World, the Entity Component System and the JOGL wrapper classes together:

Class frigo.asteroids.Game

The core Entity Component System:

Package frigo.asteroids.core

Package frigo.asteroids.message

The JOGL specific wrapper classes connecting JOGL and the game World:

Package frigo.asteroids.jogl

Asteroids specific Systems and Component

Package frigo.asteroids.component

Package frigo.asteroids.logic

Asteroids World Factory:

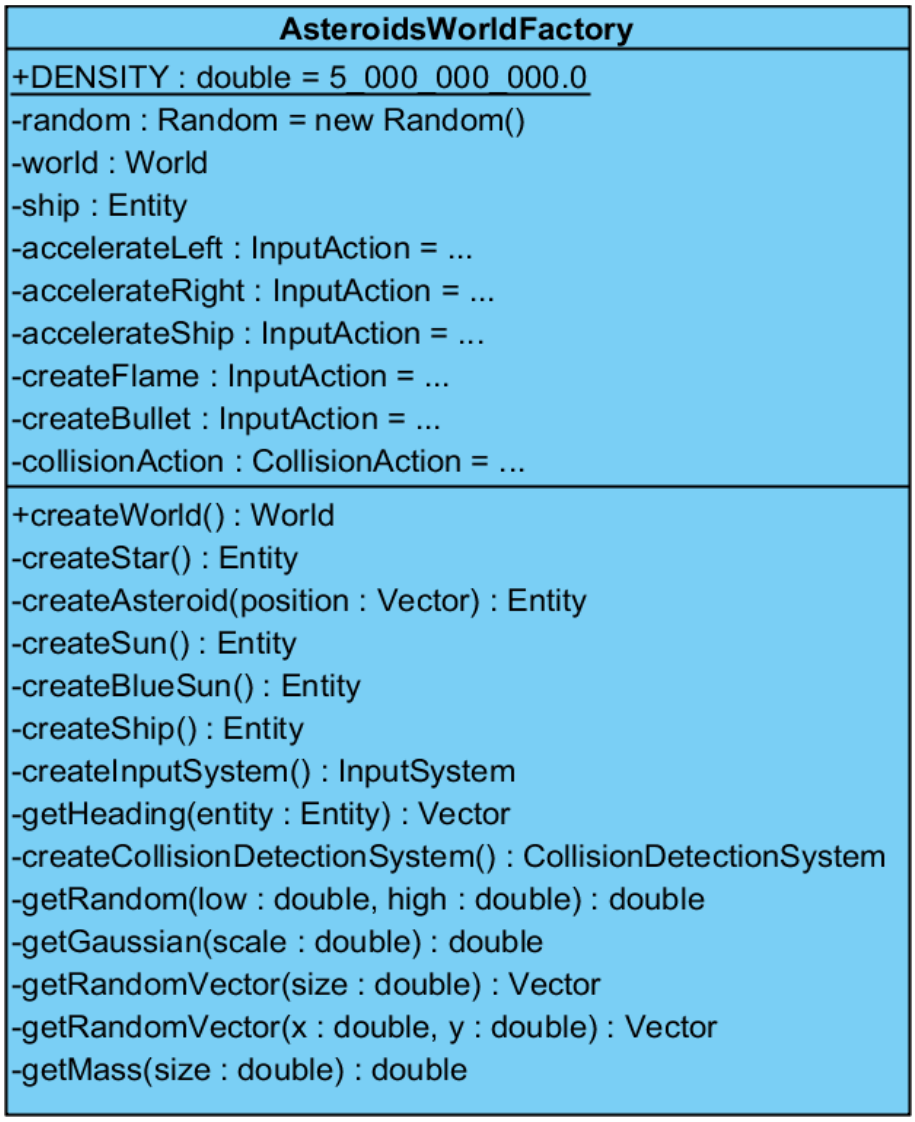
Class frigo.asteroids.AsteroidsWorldFactory

Roles of classes

Package frigo.asteroids

This package and subpackages contain all classes found in Asteroids

AsteroidsWorldFactory



This class creates a World that can be interpreted by the ECS engine and populates it with the entities proper for the Asteroids game, as well as adding functions handling keyboard inputs and collisions.

Game

Game
-world : World -keyListener : JOGLKeyListener -worldUpdater : JOGLWorldUpdater -renderer : JOGLRenderer
+main(args : String []) : void +Game() +keyPressed(e : KeyEvent) : void +keyReleased(e : KeyEvent) : void +init(drawable : GLAutoDrawable) : void +dispose(drawable : GLAutoDrawable) : void +display(drawable : GLAutoDrawable) : void +reshape(drawable : GLAutoDrawable, x : int, y : int, width : int, height : int) : void

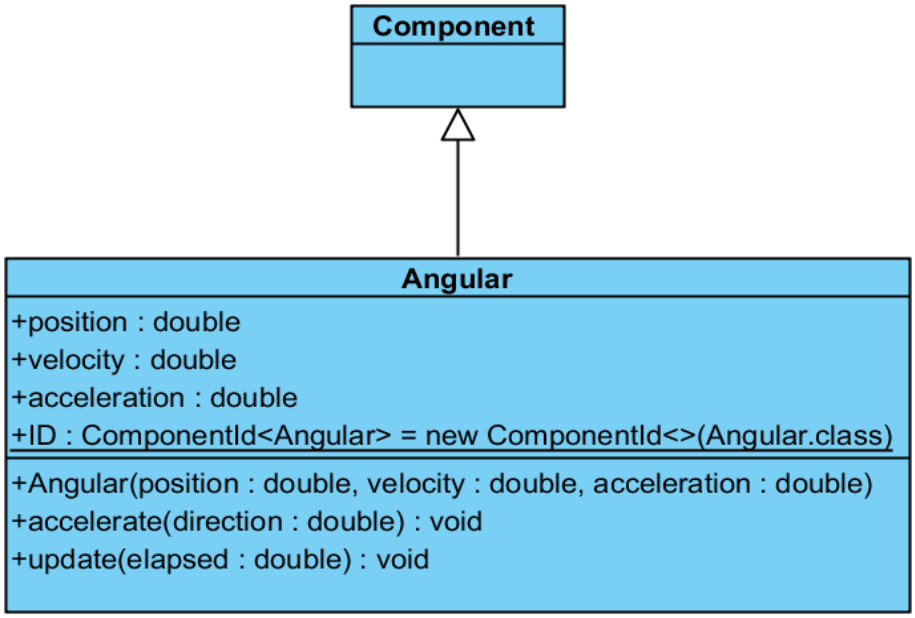
This class is the main class of the entire game. In its constructor it creates a game World via AsteroidsWorldFactory, a JOGLKeyListener, a JOGLWorldUpdater and a JOGLRenderer. In other words, its role is to constructs the game world and ties it to the external interfaces via JOGL. It implements GLEventListener and KeyListener, delegating calls for those interfaces to its fields.

In standalone application mode it spawns a JOGLRunner to run the game. In applet mode, the HTML page specifies JOGLNewtApplet1Run as the runner.

Package frigo.asteroids.component

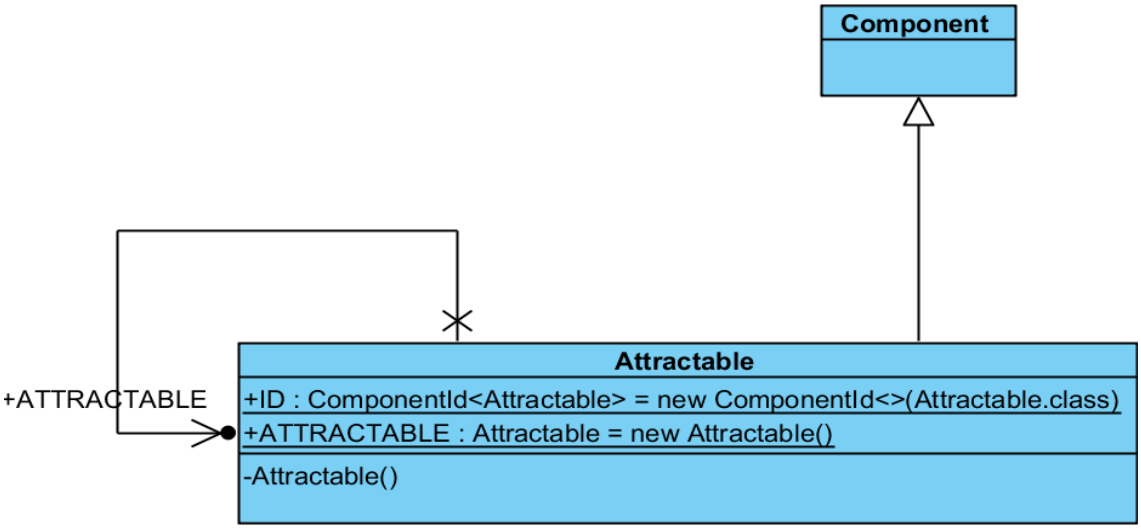
Contains the various components that can be found in the entities of the game.

Angular



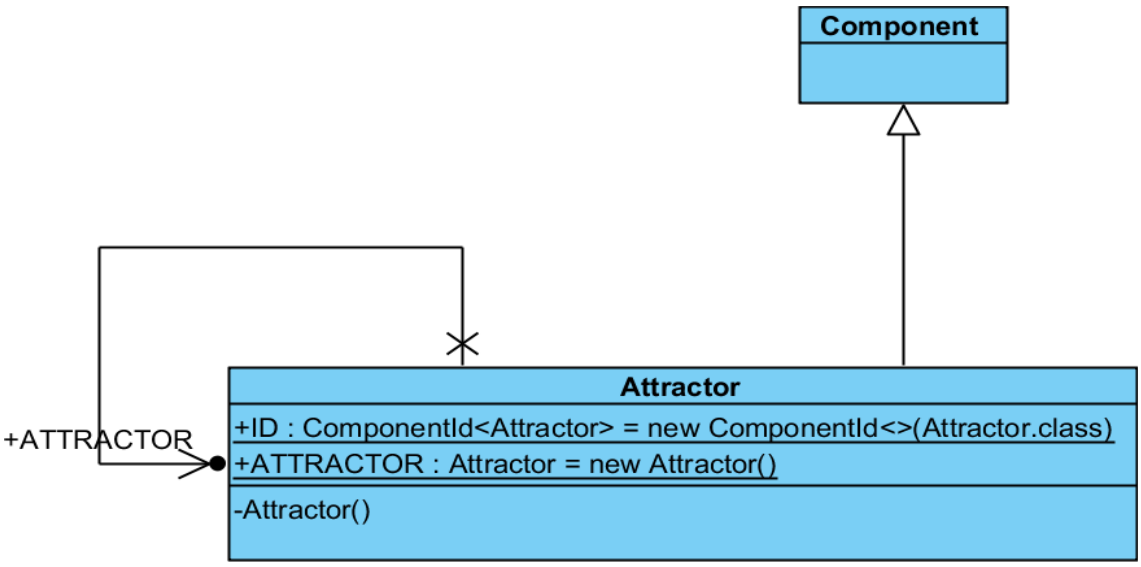
This component stores the physical properties of angular direction, velocity and acceleration. Used for asteroids, suns, the spaceship, flames, missiles and debris in the game. Entities with this component are processed by RotationSystem. Also used by JOGLRenderer.

Attractable



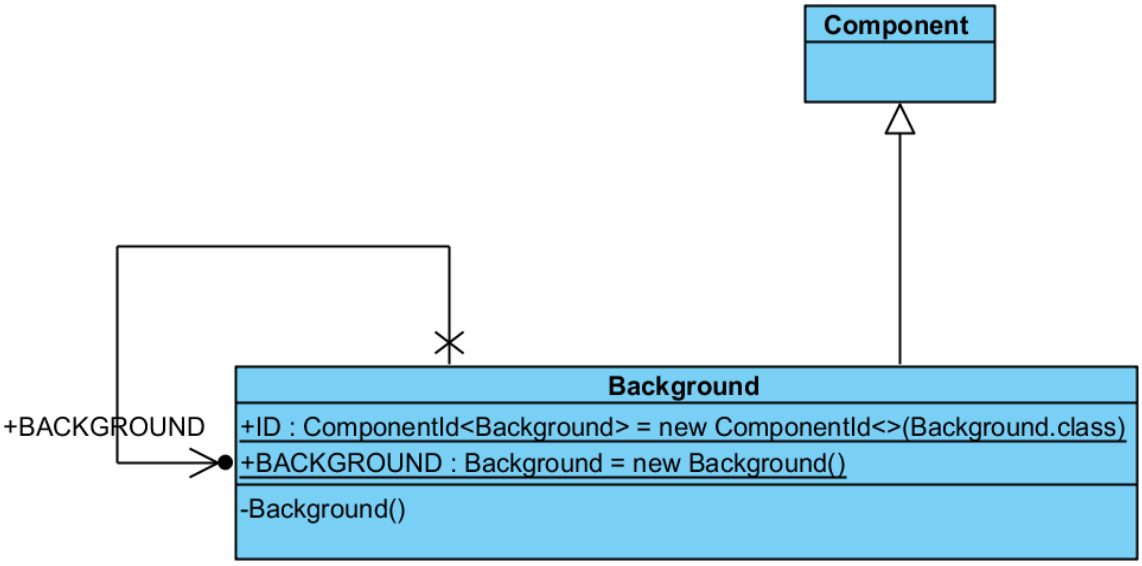
The gravitational simulation of the game requires entities subject to gravity tagged either with Attractable or Attractor components or both. Only Attractable entities are attracted by Attractor entities. Used for asteroids, the spaceship, flames, missiles and debris in the game. Entities with this component are processed by GravitySystem.

Attractor



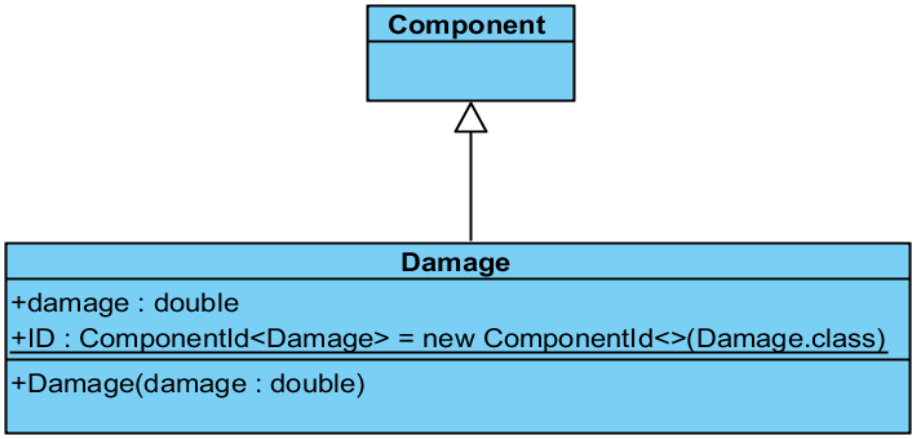
The gravitational simulation of the game requires entities subject to gravity tagged either with Attractable or Attractor components or both. Only Attractable entities are attracted by Attractor entities. Used for suns in the game. Entities with this component are processed by GravitySystem.

Background



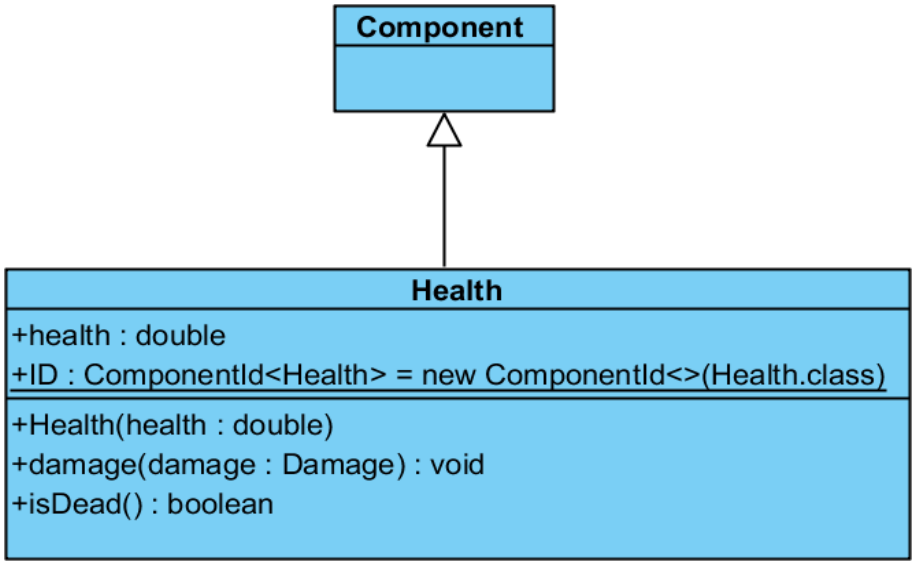
Implies that the entity should be rendered in the background, not relative to the player controlled starship. Used for backgrounds stars in the game. Used by JOGLRenderer.

Damage



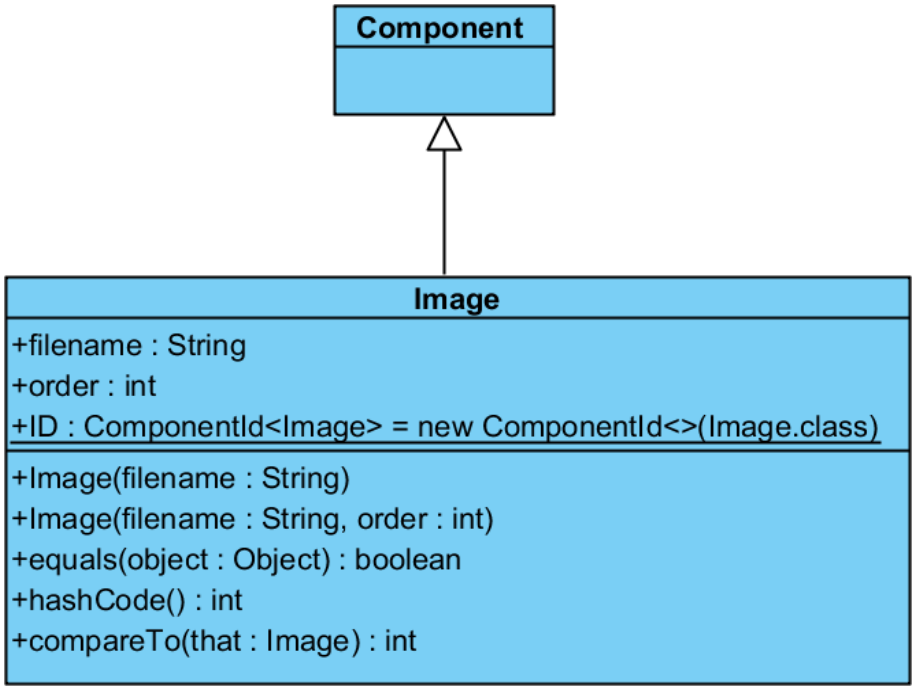
Determines how much damage an entity can deal to entities with Health component. Used for missiles in the game. Entities with this component are processed by CollisionDetectionSystem

Health



Determines how much punishment an entity can withstand before exploding. Used for asteroids and suns in the game. Entities with this component are processed by CollisionDetectionSystem.

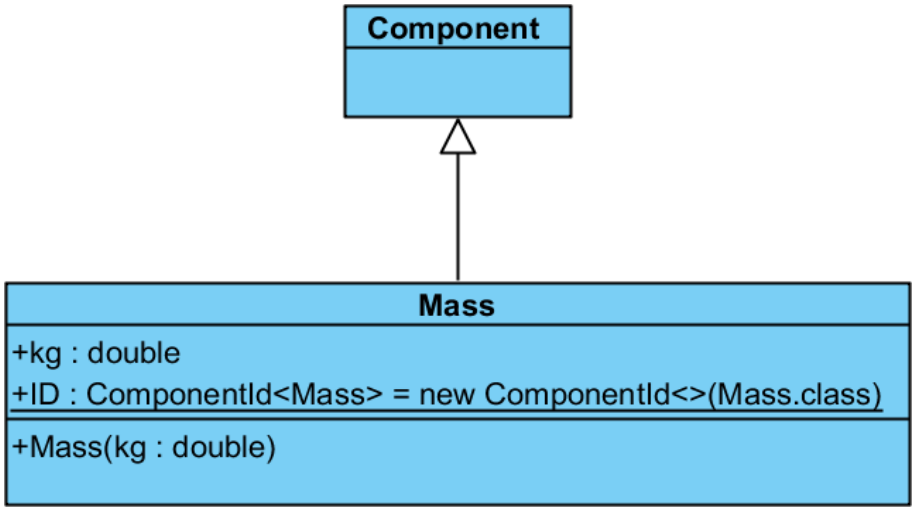
Image



Implies that an entity should be rendered with the given texture and rendering order. Used for asteroids, suns, the spaceship, flames, missiles and backround stars in

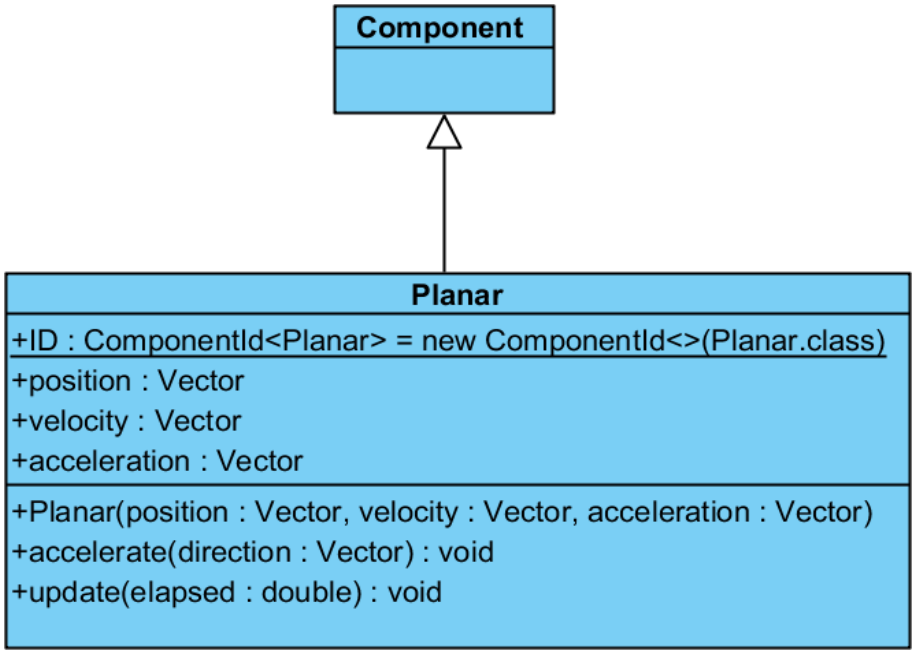
the game.

Mass



Stores the mass of an entity for purposes of gravity and explosion calculation. Used for asteroids, suns, the spaceship, flames, missiles and debris in the game. Used by GravitySystem and collision functions.

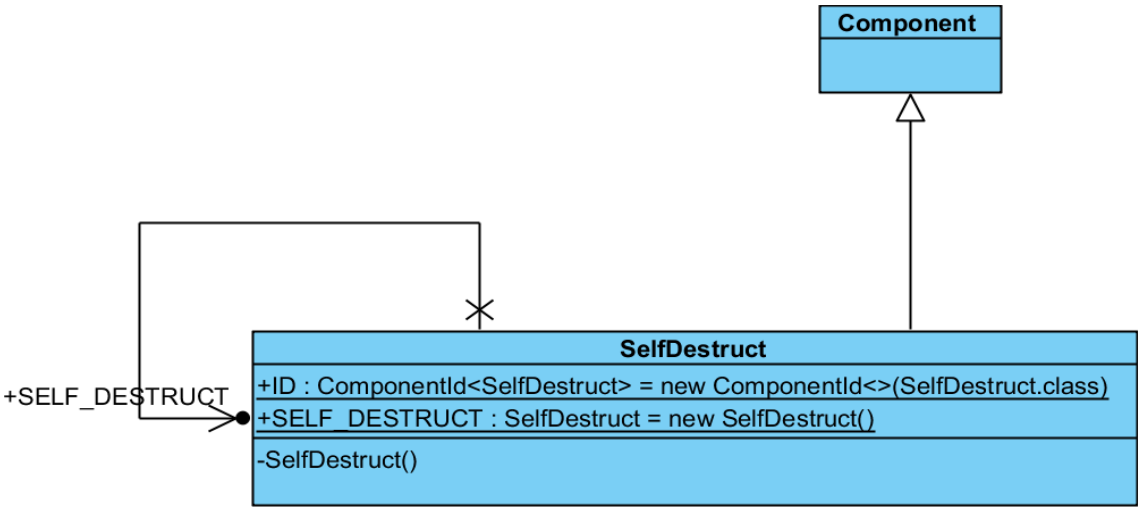
Planar



This component stores the physical properties of euclidean (planar) direction, velocity and acceleration. Used for asteroids, suns, the spaceship, flames, missiles,

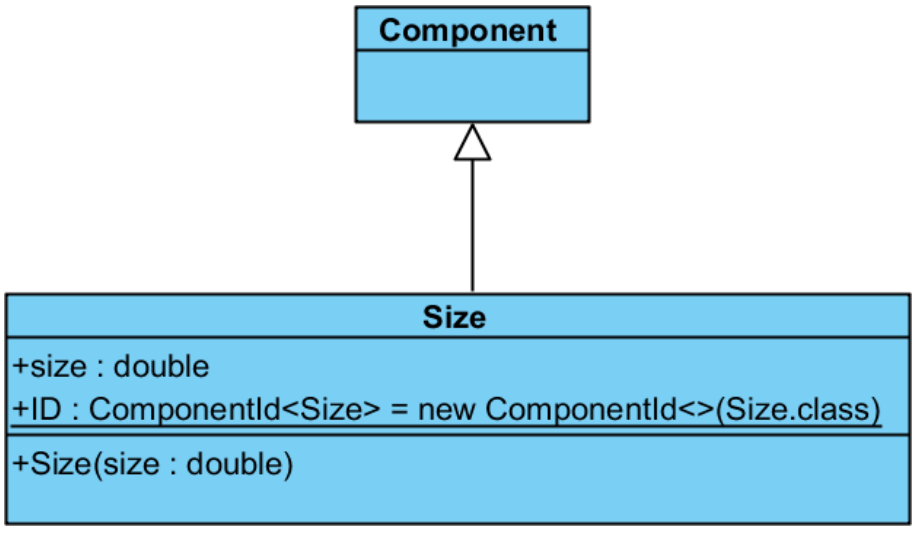
debris and background stars in the game. Entities with this component are processed by MovementSystem, GravitySystem, CollisionDetectionSystem. Also used by JOGLRenderer.

SelfDestruct



Implies that an entity should be removed. Used for flames and entities destroyed by missiles. Entities with this component are processed by SelfDestructSystem.

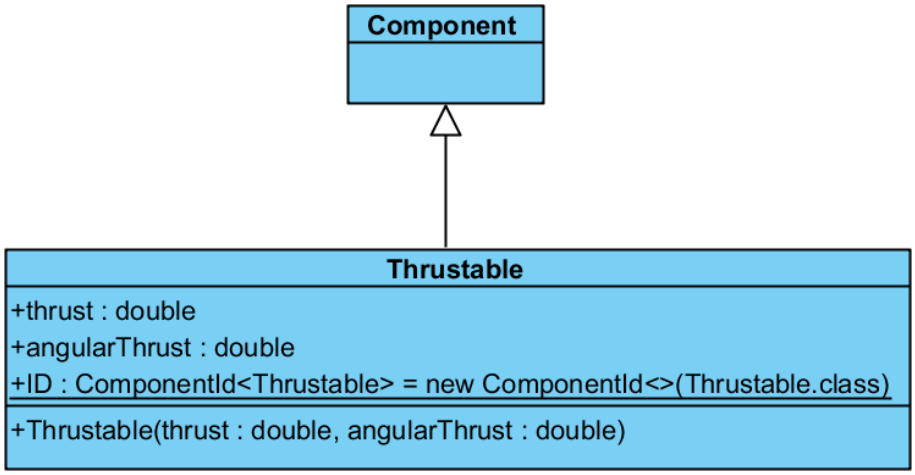
Size



Stores the physical size of the bounding box of the entity. Used for asteroids, suns, the spaceship, flames, missiles and backround stars in the game. Entities with this component are processed by the CollisionDetectionSystem. Also used by

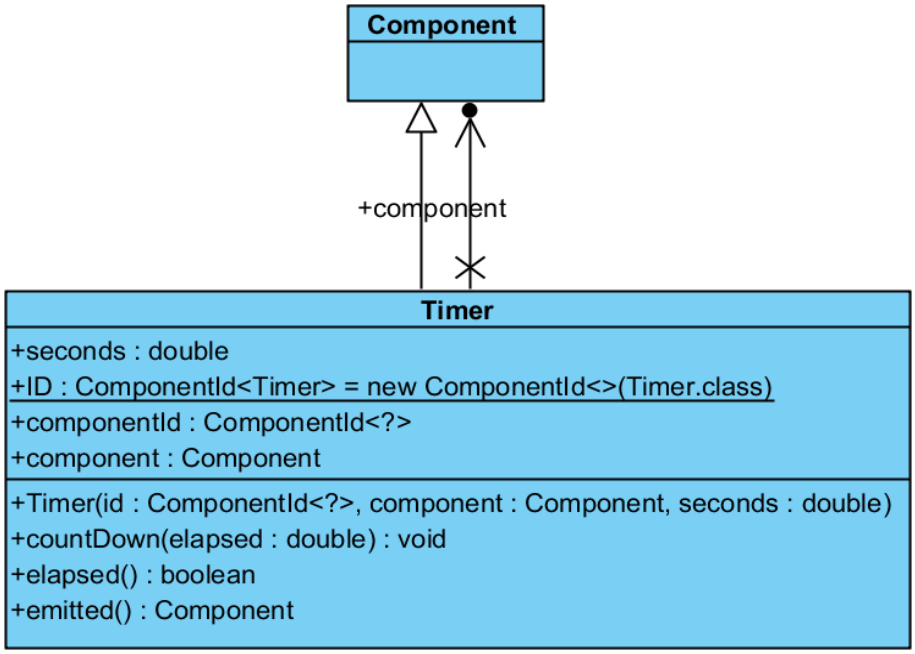
JOGLRenderer.

Thrustable



Implies that an entity containing this component can be controlled by thrusting. Also contains the amount of euclidean and angular thrust that can be achieved by doing so. Used for the spaceship in the game. Used by various input actions.

Timer



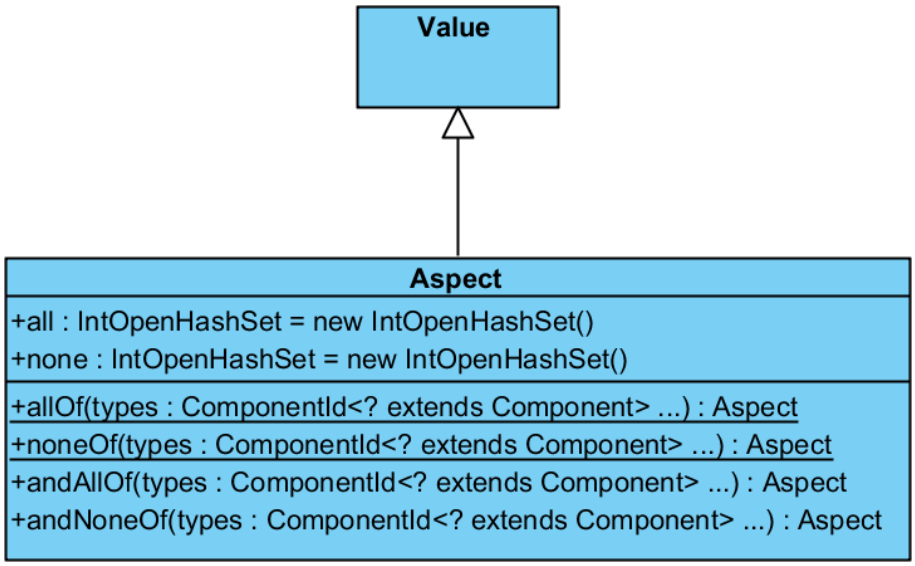
This is a component used to delay the introduction of other components into the entity. It is used to delay the self destruction of flames, bullets and debris in the game. It

is processed by TimerSystem.

Package frigo.asteroids.core

This package contains the core ECS engine built for the game.

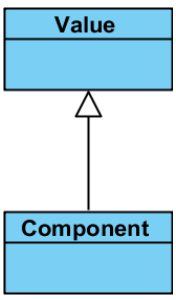
Aspect



This class is used by Systems to express the specific Component composition of entities they can process.

For example, GravitySystem works on Entities containing Attractor or Attractable, Mass, Planar, and thus uses `Aspect.allOf(Attractor.ID, Mass.ID, Planar.ID)` and `Aspect.allOf(Attractable.ID, Mass.ID, Planar.ID)` to query the appropriate entities from the game World.

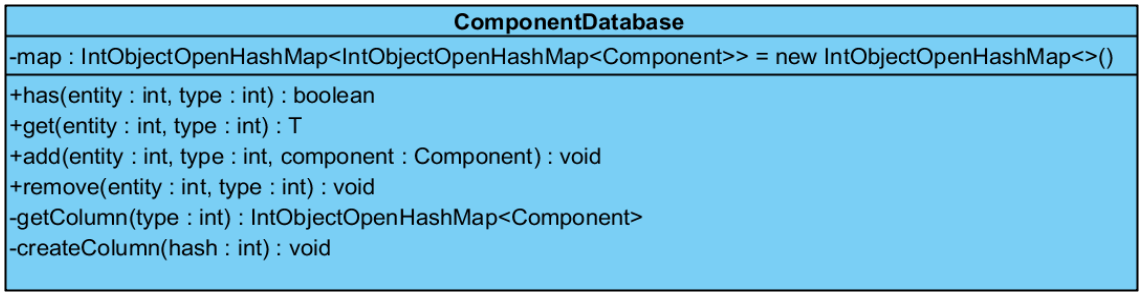
Component



All component classes should derive from this class. All classes working with

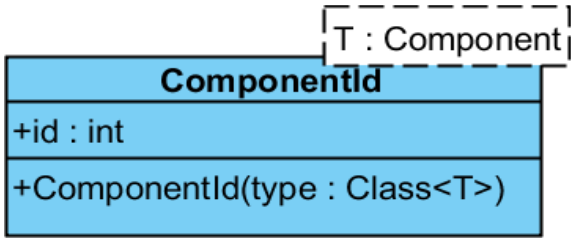
components have this class as a constraint.

ComponentDatabase



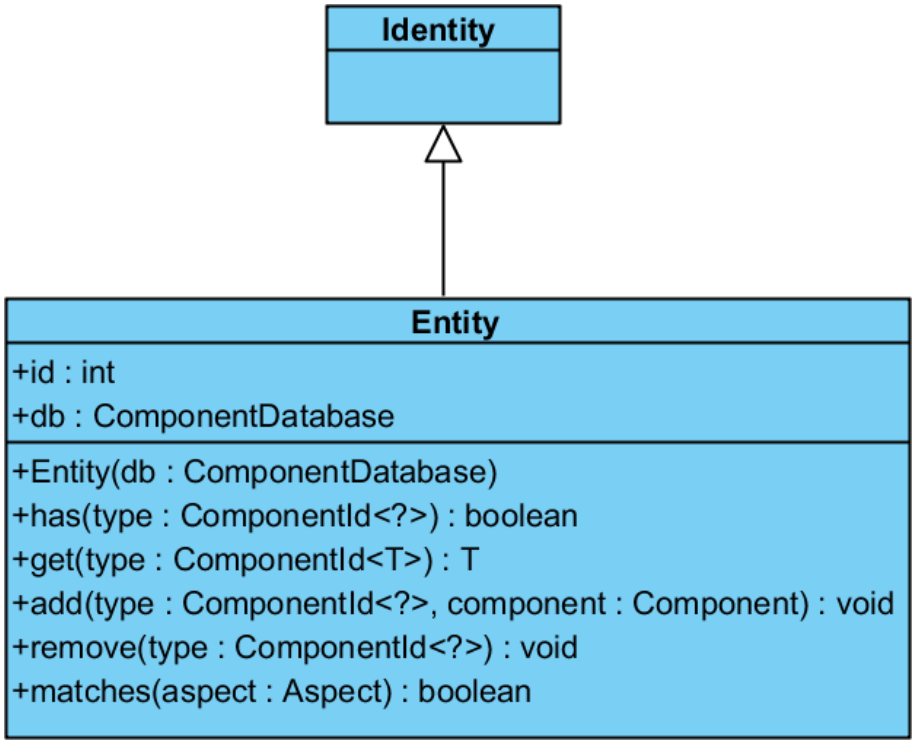
This class stores all the components found in the entities in the game. It stores entities of the same class contiguously, rather than collecting them under entities. This is in line with the Data-Driven Programming perspective of Entity Component Systems.

ComponentId



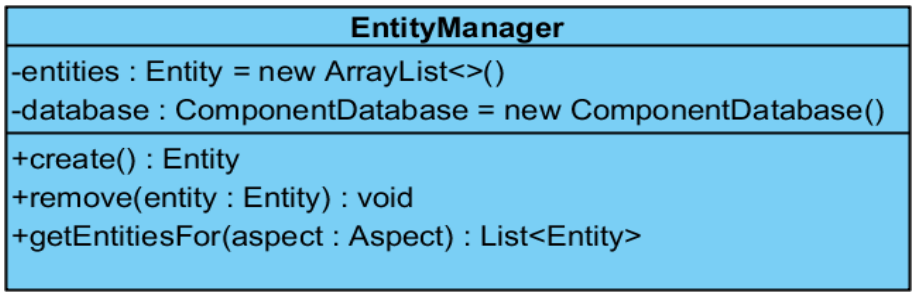
This class is a wrapper for a unique integer identifier for Component classes. It is used in place of `Class<? extends Component>` objects to provide better performance and access to specialized primitive containers like those that found in Trove.

Entity



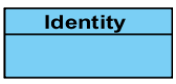
This class corresponds to Entity in Entity Component Systems. In theory it is a set of Components. In practice it is usually just a unique identifier, with the Components stored elsewhere. The Asteroids implementation follows this custom, it contains a unique identifier and a delegates to ComponentDatabase.

EntityManager



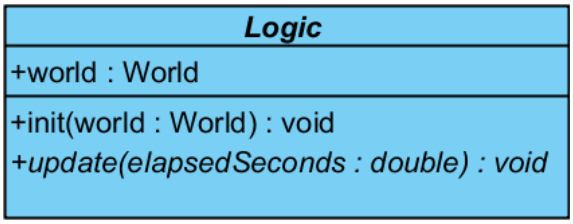
Stores the list of Entities currently in the game World.

Identity



Implies that instances of a class are identities rather than values. Entity derives from Identity.

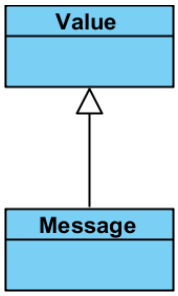
Logic



This class corresponds to System in Entity Component Systems. All Systems derive from this class: CollisionDetectionSystem, GravitySystem, InputSystem, MovementSystem, RotationSystem, SelfDestructSystem, TimerSystem.

It is named Logic rather than System to prevent confusion with the built-in Java API class java.lang.System which is automatically included in all Java classes.

Message

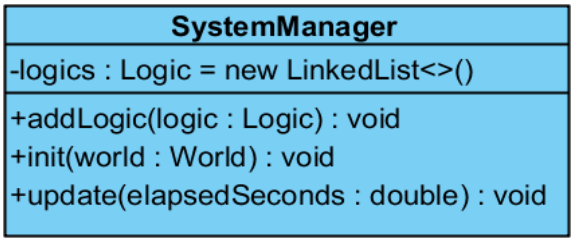


Implies that instances of a class are messages capable of being sent across architectural boundaries like networks. Keyboard input messages derive from it.

MessageManager

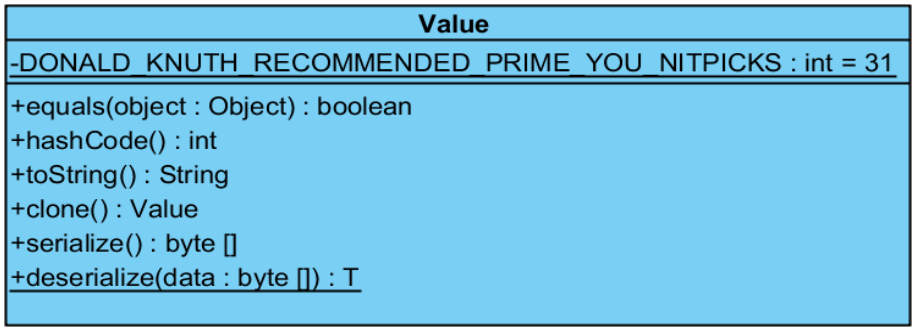
Stores messages currently active in the game World.

SystemManager



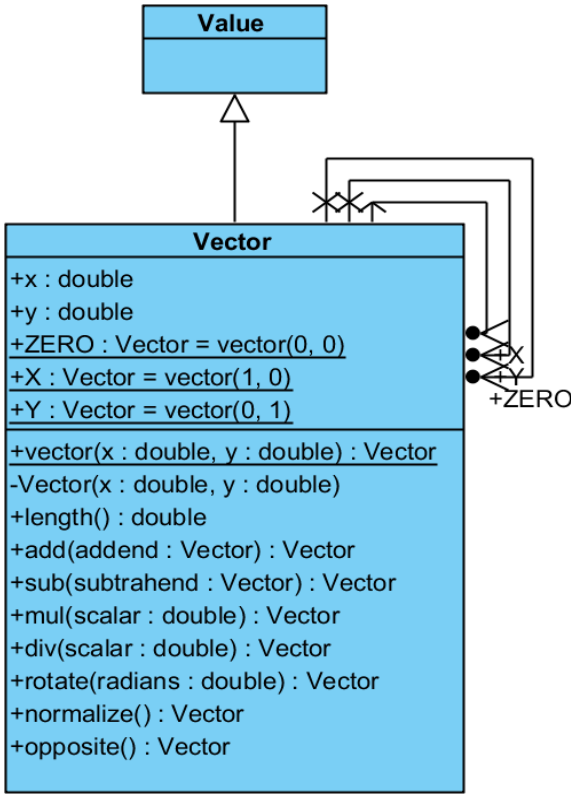
Stores the Systems registered in the game World.

Value



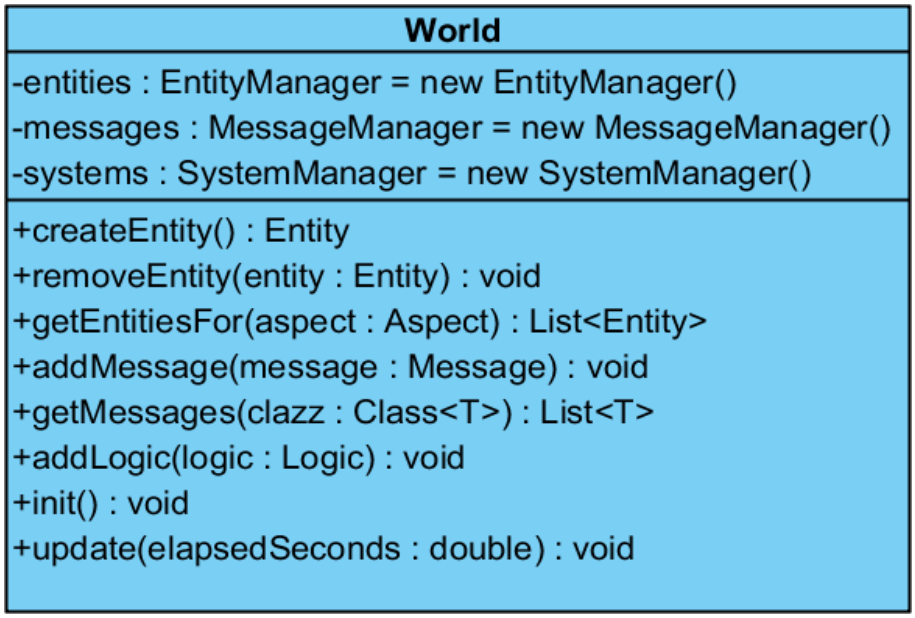
Implies that instances of a class are values rather than identities. Aspect, Components, Messages and Vector derives from it.

Vector



Two dimensional Vector implementation. Used in the Planar component and transitively in systems using the Planar component: GravitySystem, MovementSystem, CollisionDetectionSystem. Also used in JOGLRenderer.

World



Primary interface and container for the game world and its current state. Provides facades and delegations to EntityManager, MessageManager and SystemManager. Created by AsteroidWorldFactory. Used by all JOGL glue classes and all systems.

Package frigo.asteroids.jogl

JOGLKeyListener

JOGLKeyListener
-keyEvents : LinkedBlockingQueue<KeyEvent> = new LinkedBlockingQueue<>() -held : Map<Short, KeyEvent> = new HashMap<>() -world : World
+JOGLKeyListener(world : World) +keyPressed(event : KeyEvent) : void +keyReleased(event : KeyEvent) : void +init(drawable : GLAutoDrawable) : void +display(drawable : GLAutoDrawable) : void +dispose(drawable : GLAutoDrawable) : void +reshape(drawable : GLAutoDrawable, x : int, y : int, width : int, height : int) : void

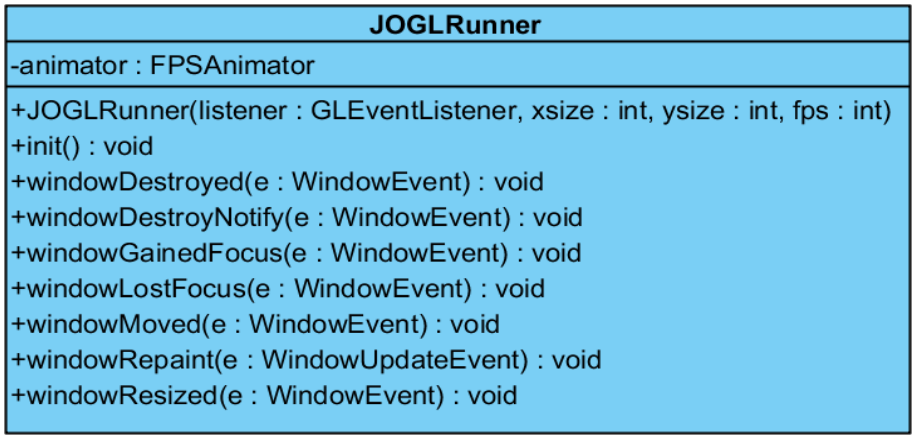
Provides a layer between JOGL and the game World for purposes of keyboard input interpretation. Translates between JOGL KeyEvent callback events and in-game KeyPressed, KeyHeld, KeyReleased events.

JOGLRenderer

JOGLRenderer
-world : World -textures : TextureBuffer = new TextureBuffer() -player : Entity
+JOGLRenderer(world : World) -getPlayer() : Entity +init(drawable : GLAutoDrawable) : void +display(drawable : GLAutoDrawable) : void -drawEntities(gl : GL2, entities : List<Entity>) : void -separateEntitiesByImage(entities : List<Entity>) : Map<Image, List<Entity>> -drawEntitiesByImage(gl : GL2, Image : Image, entities : List<Entity>) : void -addVertex(textureBuffer : DoubleBuffer, vertexBuffer : DoubleBuffer, entity : Entity, u : double, v : double) : void -textureSpaceToVertexSpace(entity : Entity, u : double, v : double) : Vector -getAngularPosition(entity : Entity) : double +dispose(drawable : GLAutoDrawable) : void +reshape(drawable : GLAutoDrawable, x : int, y : int, width : int, height : int) : void

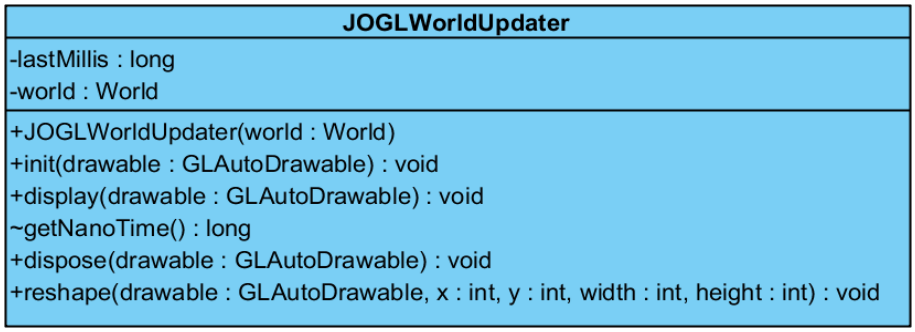
OpenGL renderer for the game. All rendering is done relative to the position of the starship of the player. Only entities corresponding to `Aspect.allOf(Planar.ID, Size.ID, Image.ID)` are rendered. Entities are rendered in the order specified by their Image components. Within the same order, entities with the same texture are rendered at the same time to avoid the cost of texture binding. The renderer respects angular positions and handles aspect ratio changes correctly.

JOGLRunner



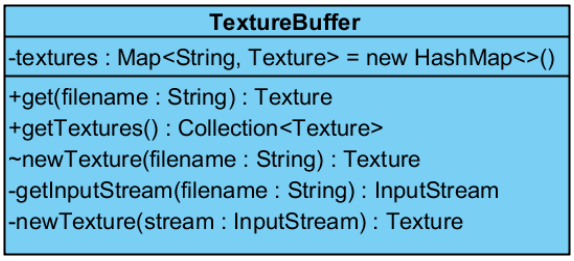
This class handles the creation of an OpenGL surface represented by a GLWindow instance, connecting it with the main Game class, and starting the animation callbacks by creating an FPSAnimator class. Used only in standalone mode, a builtin JOGL class called JOGLNewtApplet1Run handles these tasks in applet mode.

JOGLWorldUpdater



This class is responsible for updating the game World in response to callbacks by FPSAnimator. It also keeps track of the time elapsed since the last update.

TextureBuffer

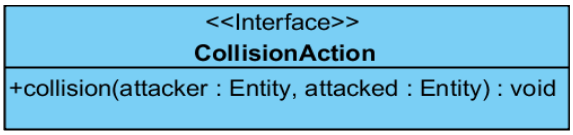


Keeps a buffer of loaded textures in the memory to avoid excessive disk loads.

Subpackages of frigo.asteroids.logic

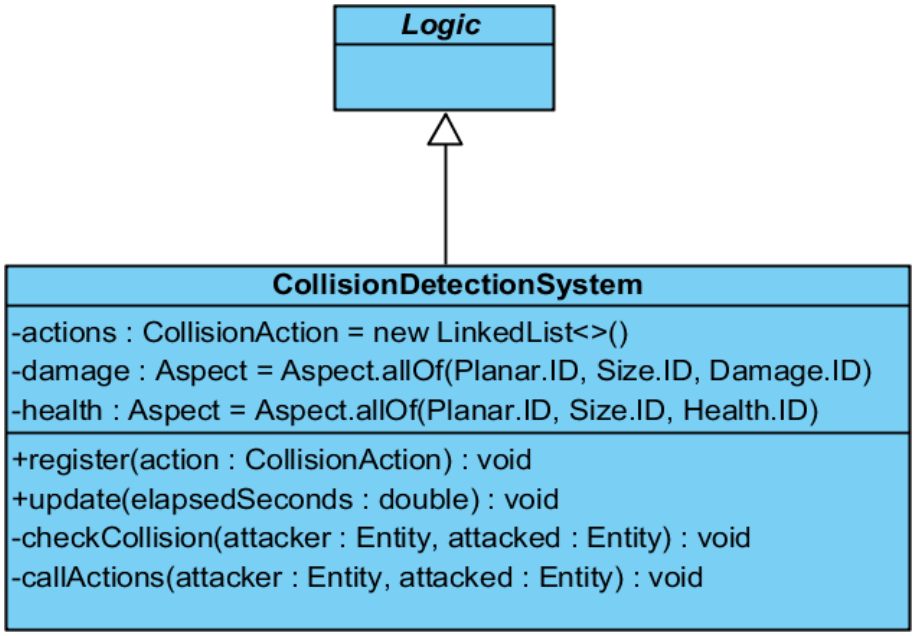
These package and subpackages contain all Systems used by game and their required classes and interfaces.

CollisionAction



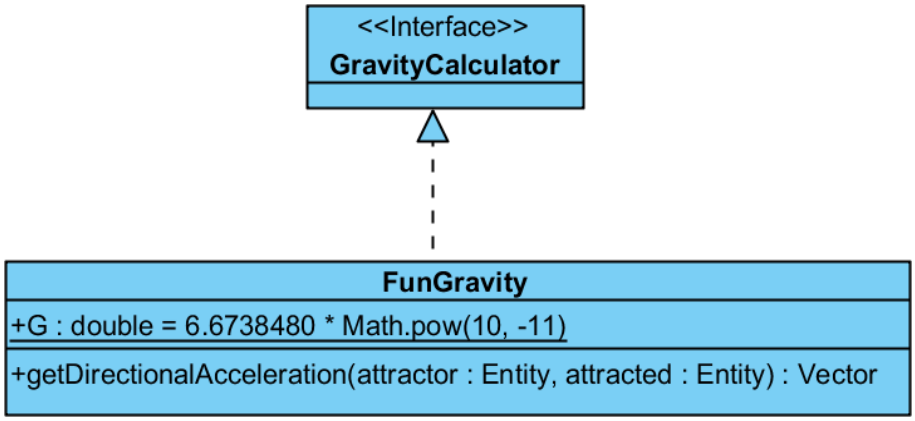
Interface of collision callback action accepted by CollisionDetectionSystem.

CollisionDetectionSystem



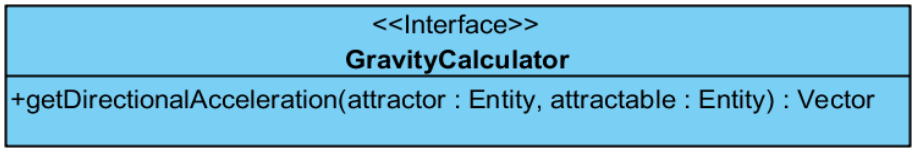
Detects collisions between entities conforming to `Aspect.allOf(Planar.ID, Size.ID, Damage.ID)` and `Aspect.allOf(Planar.ID, Size.ID, Health.ID)` and calls the registered collision callback events.

FunGravity



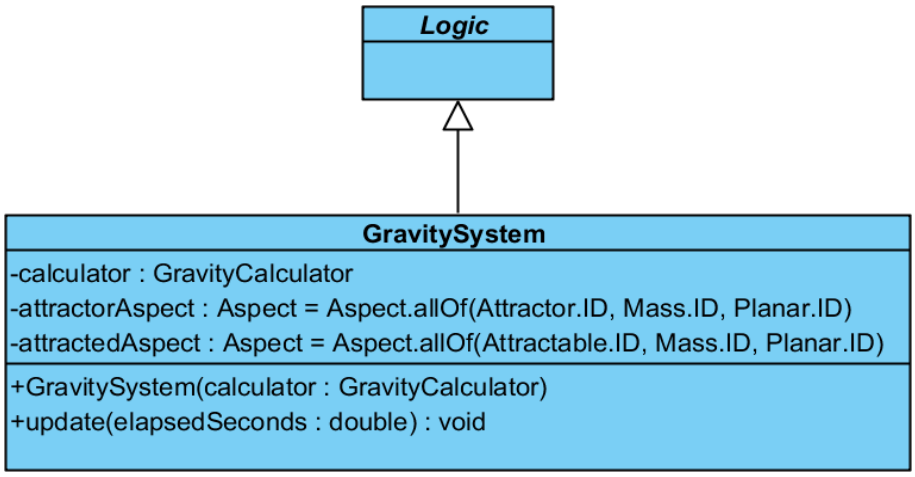
A variant of the pairwise Newtonian gravitation calculation formula. The force between two entities is calculated by dividing with the distance rather than the square of the distance. Used during development to explore options regarding gravity. Produced a spectacular sphere like explosion when gravity was applied to background stars.

GravityCalculator



Interface implemented by FunGravity and NewtonianGravity. Injected into GravitySystem. Used during development to explore options regarding gravity.

GravitySystem

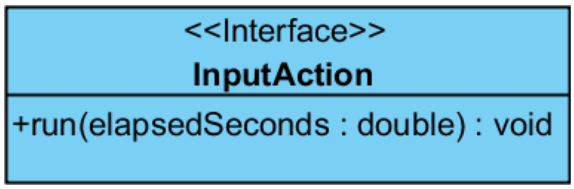


Calculates pairwise gravitation subject to all entities subject to `Aspect.allOf(Attractor.ID, Mass.ID, Planar.ID)` and `Aspect.allOf(Attractable.ID, Mass.ID, Planar.ID)`, then applies it to their acceleration.

NewtonianGravity

Pairwaise Newtonian gravitation calculation formula. A straightforward implementation of the $F = G\frac{m_1 \dot{m}_2}{r^2}$ formula.

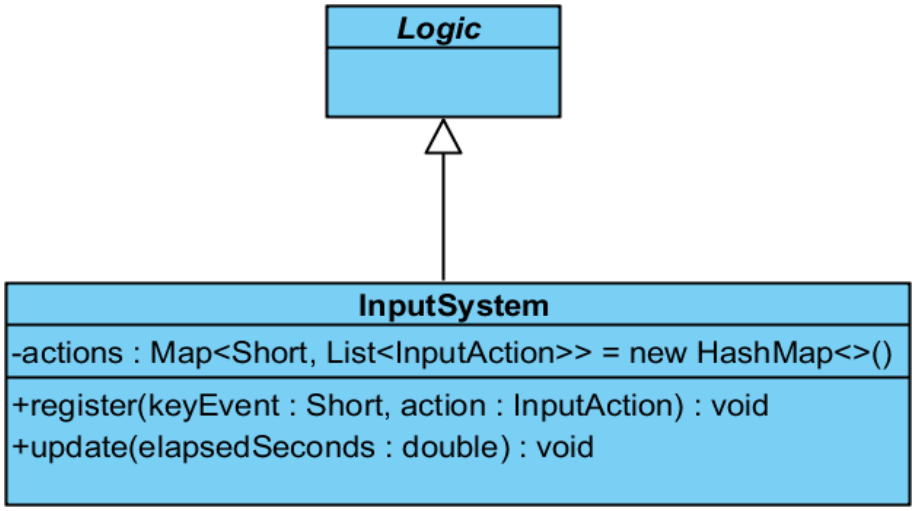
InputAction



Callback interface implemented by keyboard input event callbacks registered during game World creation. The InputAction instances used in the game:

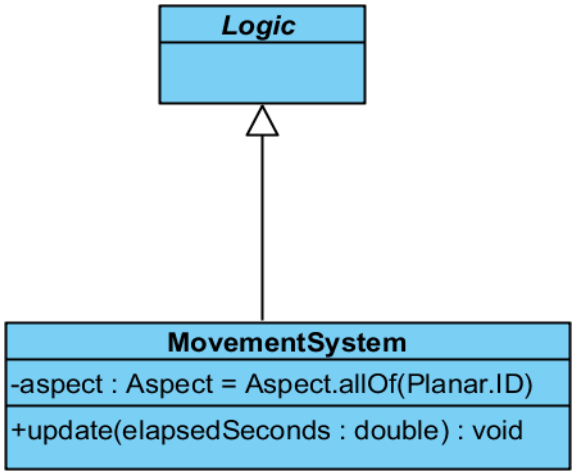
- Left key binding: counterclockwise angular acceleration
- Right key binding: clockwise angular acceleration
- Up key binding: forward acceleration
- Up key binding: exhaust flame creation
- Space key binding: firing of BFM-9001 missiles

InputSystem



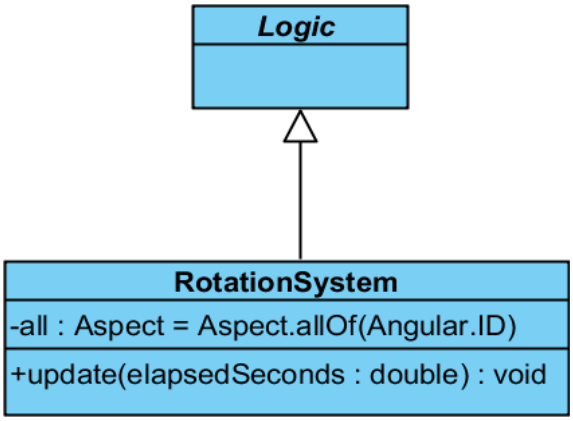
Calls registered InputAction callbacks on associated keyboard input events.

MovementSystem



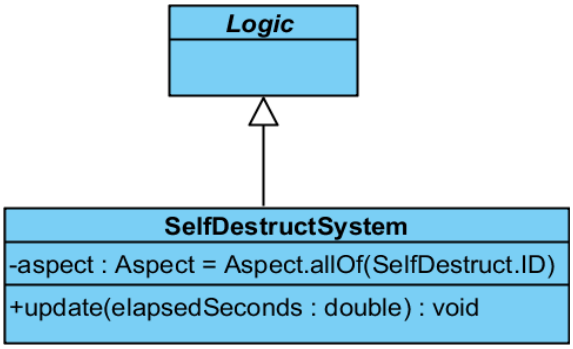
Updates the Planar properties position, velocity and acceleration of all entities conforming to `Aspect.allOf(Planar.ID)` based on their acceleration.

RotationSystem



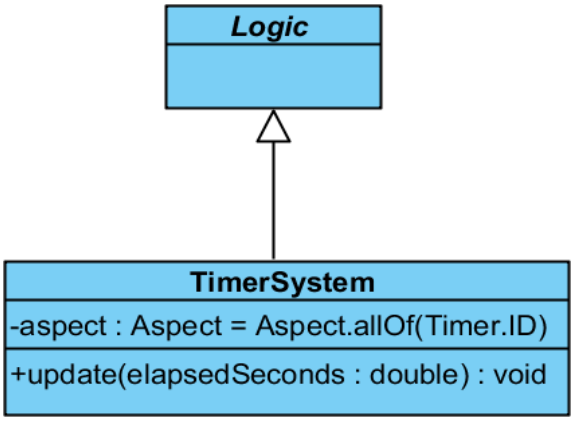
Updates the Angular properties angular position, angular velocity and angular acceleration of all entities conforming to `Aspect.allOf(Angular.ID)` based on their angular acceleration.

SelfDestructSystem



Deletes entities with SelfDestruct component. Usually done at the end of the game loop.

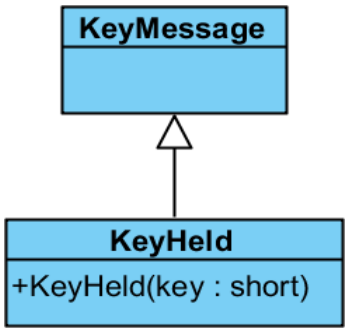
TimerSystem



Used to delay the introduction of other components into an entity. It is used to delay the self destruction of flames, bullets and debris in the game. Works on entities conforming to `Aspect.allOf(Timer.ID)`.

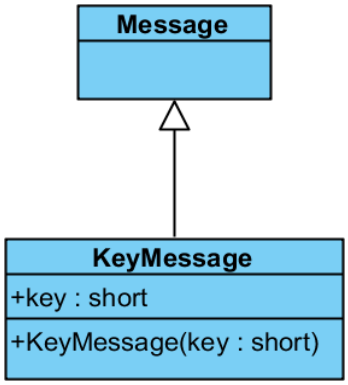
Package frigo.asteroids.message

KeyHeld



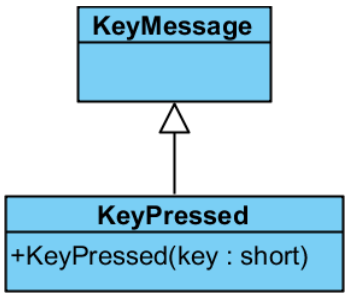
Message emitted by JOGLKeyListener to signal that a key is being held since last update. At the moment there is no difference in handling KeyHeld and KeyPressed in the game.

KeyMessage



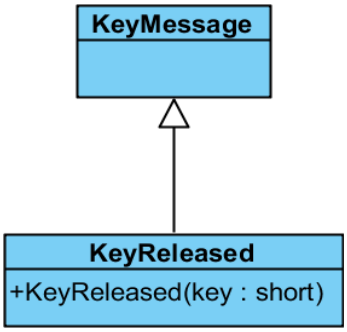
Superclass of keyboard input event messages.

KeyPressed



Message emitted by JOGLKeyListener to signal that a key was pressed since last update. At the moment there is no difference in handling KeyHeld and KeyPressed in the game.

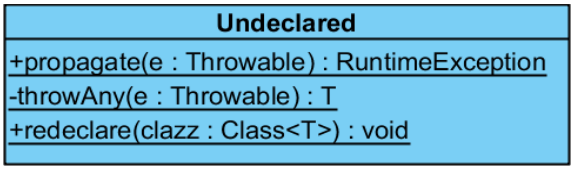
KeyReleased



Message emitted by JOGLKeyListener to signal that a key was released since last update. At the moment there is no InputAction handling of KeyReleased messages in the game.

Package frigo.asteroids.util

Undeclared



Utility class to work around one of the most controversial features of Java, namely that of checked exceptions. With the help of this class it is possible to throw a checked exception where it is not explicitly declared in the throws clause of the method declaration, and only redeclaring on the level where we intend to catch it.

Known bugs

Unstable Solar System

The solar system does not seem to be stable for unknown reasons. After several minutes, the asteroids start to drift further and further away from the two stars and the gravitational center of the system. The author believes the initial random configuration of the solar system does not satisfy some unfamiliar constraint of a stable solar system.

Applet performance

Possibly due to the extra security needed to run Java Applets, the applet version of the game run by JOGL's JOGLNewtApplet1Run applet runner class suffers from performance problems, providing about one magnitude less performance when measuring by frames per second with all other variables unchanged.

Partly because of repeated unsuccessful attempts to diagnose and fix the root cause of this low performance of applets, and partly because of the difficulty on the side of the user to run the applet safely, the applet feature will not be pursued in this project, and given the opportunity, it will be removed.

Possibilities for future development

Earth

Since the upkeep of Arthur's Mantle is so expensive, it can be activated only for short periods of time, during which the Captain has to destroy an incoming wave of asteroids.

Should the Captain run out of time, the Earth reappers in the middle of the chaos and is highly vulnerable to any collision - be it from asteroids or BFM-9001 missiles.

Thus Earth would serve as a gameplay element, a backstory based penalty for slow gameplay, alternative to a hard cutoff timer, more suited to maintain the illusion of an integral in-game universe.

Sun

Should the player accidentally blow up the sun, the game should end for obvious reasons.

Color of the starship

Beta testers often complained that sometimes the starship is barely visible among the similarly colored asteroids.

Possible solutions for this complaint involve either changing the color of the starship, adding different colored stripes, or implementing a selection/highlight system as part of the graphical user interface. This latter solution would be vastly preferred, for this game and other games in general, especially strategy games.

Less confusing controls

Beta testers also complained that there is no feedback for the angular acceleration of the starship, making the relatively simple realistic controls of the game more confusing than they should be.

A possible solution would be to add small side exhaust flames when the user is

accelerating the angular rotation of the ship in either direction.

Display count of asteroids

It would improve player experience if the number of remaining asteroids was displayed.

Game states

A quite logical possibility of improvement to the current revision of the game is to introduce game states. Game states are states that are applicable to the entire game, rather than a single entity or a group of entities. For example, *pause* is a game state, the amount of *health* of a unit is an entity state.

There could be a lot of game states possible, the ones that would make sense for Asteroids are the following: menu, pre-game, game, pause, about to change levels, win state, loss state, et cetera.

Levels

The game could be split into waves of asteroids comparable to levels in other games. This would nicely fit in the in-universe backstory of the game.

Modern OpenGL

The game is currently using the OpenGL 2.0 interface, including glBegin() and glEnd() that has been deprecated in OpenGL 3.0 in favor of Vertex Buffer Objects where the client software transfers all vertex or texture coordinates in bulk arrays to the video card for rendering.

A future improvement could introduce modern OpenGL best practices into the rendering subsystem.

Multiplayer

A multiplayer game requires a vastly different software architecture, and networking introduces a completely different level of programming complexity that is

not present in single player games: server-client or peer-to-peer architecture, authoritative server or consensus networks, database synchronization and all theoretical problems with it, networking code with special emphasis on error handling of network failures, more heavy separation of input, game logic, database and rendering, proper abstractions in the forms of actions and responses, security issues, protection against hackers, server maintenance and associated costs, upgrade problems, just to name the major issues usually encountered during multiplayer game development.

Moreover, multiplayer games have to be designed from ground up with multiplayer features in mind. Multiplayer features influence architecture, abstractions, game logic and the development process on a fundamental level. Therefore it is prohibitively difficult to transform an already existing single player game into a networked game due to significant differences between architecture, abstractions, game logic and development process.

Since Asteroids was not meant to be a networked game, therefore it would be counterproductive to refactor it to be one. However the experience gained during its development would certainly help the development of a similar multiplayer game.

Applet

While the game runs as a Java Applet at the moment, this is an experimental feature, not reliable, and most likely will not be supported in the future. The main issue is that the performance of the applet is several times lower than the standaline application, probably due to security restrictions. Also, said security restrictions can prevent the inclusion of some libraries that are relying on elevated rights for some of their operations.