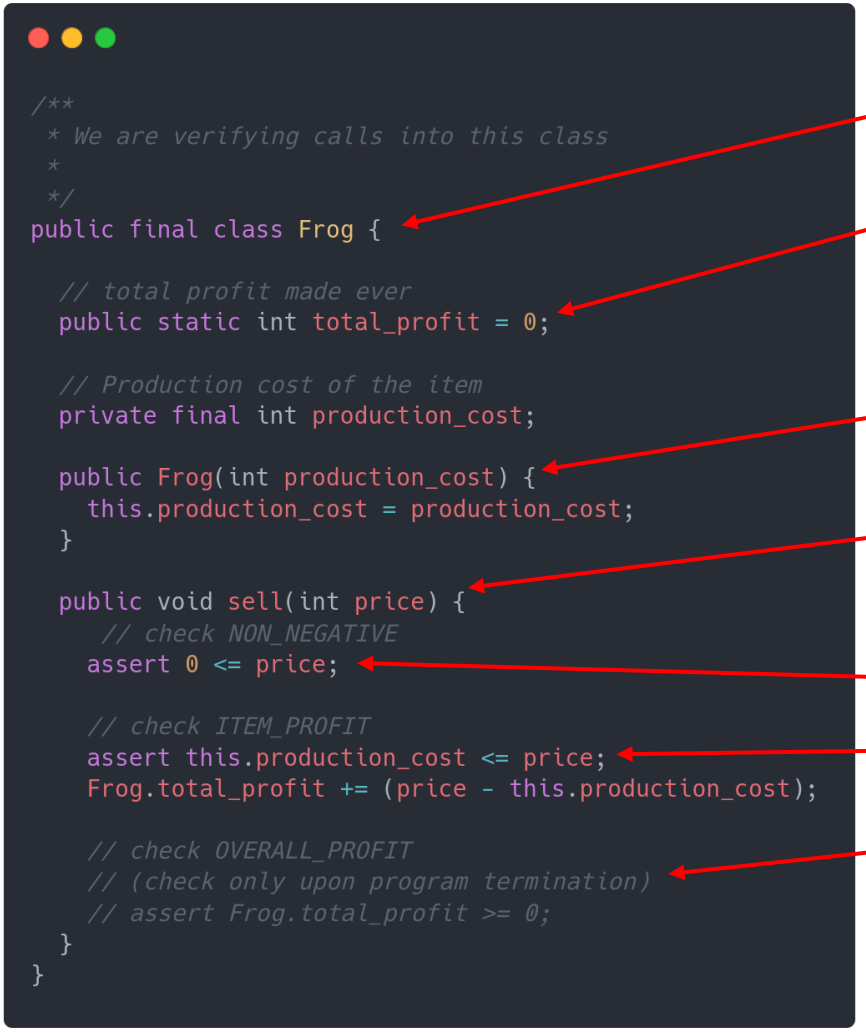
RSE – Project

2024

What is it about?



```java
/**
 * We are verifying calls into this class
 *
 */
public final class Frog {

  // total profit made ever
  public static int total_profit = 0;

  // Production cost of the item
  private final int production_cost;

  public Frog(int production_cost) {
    this.production_cost = production_cost;
  }

  public void sell(int price) {
     // check NON_NEGATIVE
    assert 0 <= price;

    // check ITEM_PROFIT
    assert this.production_cost <= price;
    Frog.total_profit += (price - this.production_cost);

    // check OVERALL_PROFIT
    // (check only upon program termination)
    // assert Frog.total_profit >= 0;
  }
}
```

Class we verify on

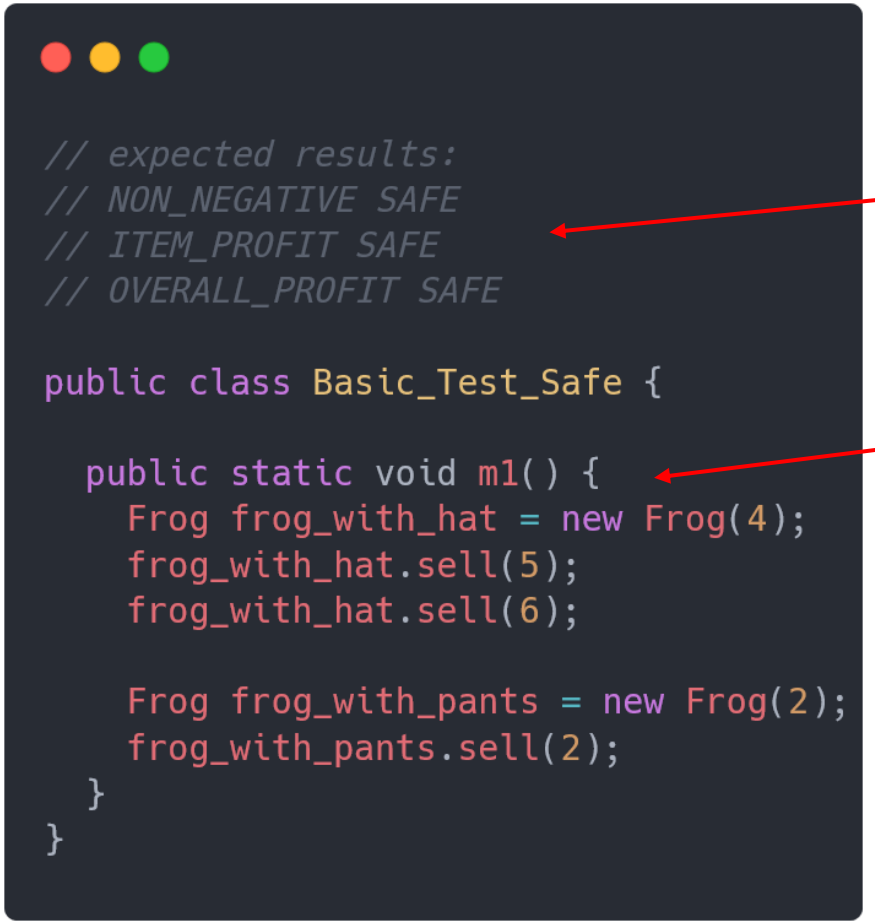
Profit from all frogs sold

Different Frogs have different prices

We can sell the same class for various prices

Three properties we would like to verify:

1. Positive price
2. No loss at sale
3. No total loss


// expected results:
// NON_NEGATIVE SAFE
// ITEM_PROFIT SAFE
// OVERALL_PROFIT SAFE
public class Basic_Test_Safe {
public static void m1() {
Frog frog_with_hat = new Frog(4);
frog_with_hat.sell(5);
frog_with_hat.sell(6);
Frog frog_with_pants = new Frog(2);
frog_with_pants.sell(2);
}
}
Expected Test Results
Analyzed Function

```
// expected results:
// NON_NEGATIVE UNSAFE
// ITEM_PROFIT UNSAFE
// OVERALL_PROFIT UNSAFE

public class Basic_Test_Unsafe {

  public void m2(int j) {
    Frog frog_with_sweater = new Frog(2);
    if(-1 <= j && j <= 3)
      frog_with_sweater.sell(j);
  }
}
```

Potentially Unsafe call

Frameworks (Analysis)

APRON (for numerical analysis)

```
{0 <= x <= 3}
    Assign(y, x + 2)
{0 <= x <= 3, 2 <= y <= 5}
```

Soot

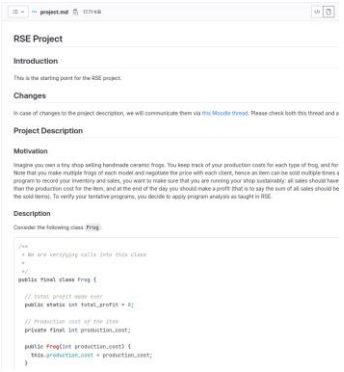
1. For parsing Java code: if => JifStmt.getCondition()
2. For pointer analysis

Frameworks (Software Engineering)

- Docker ← Environment
- GitLab CI/CD ← Continuous Integration
- JaCoCo ← Test-Coverage
- Maven ← Build System
- SLF4J/Logback ← Logging

(That's a lot of) Frameworks

Getting familiar with these is part of the assignment **(spending time there is normal and expected)**



RSE Project

Introduction

This is the starting point for the RSE project.

Changes

Project Description

Motivation

Description

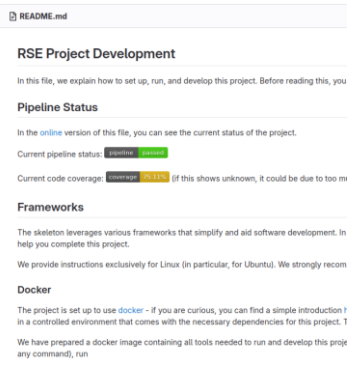
```
/**
 * We are verifying calls into this class
 *
 */
public final class Frog {

  // Total profit made ever
  public static int total_profit = 0;

  // Production cost of the item
  private final int production_cost;

  public Frog(int production_cost) {
    this.production_cost = production_cost;
  }
```

We provide resources in the project description



README.md

RSE Project Development

Pipeline Status

In the online version of this file, you can see the current status of the project.

Current pipeline status: pipeline passed

Frameworks

The skeleton leverages various frameworks that simplify and aid software development. In
help you complete this project.

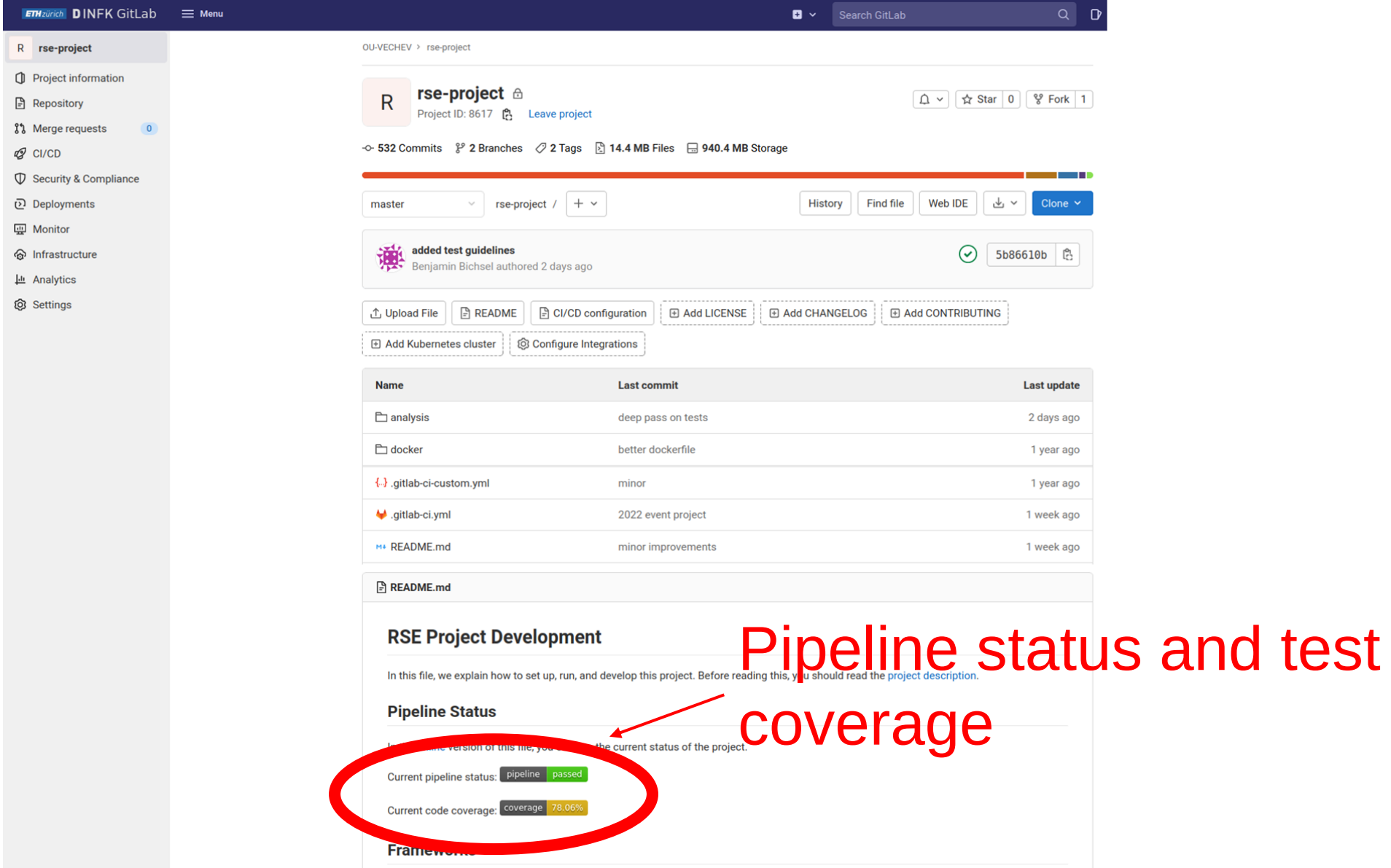
We provide instructions exclusively for Linux (in particular, for Ubuntu). We strongly recom

Docker

The project is set up to use docker - if you are curious, you can find a simple introduction
in a controlled environment that comes with the necessary dependencies for this project.

We have prepared a docker image containing all tools needed to run and develop this proje
any command), run

And installation instructions in README


ETH zürich DINFK GitLab
Menu
Search GitLab
R rse-project
Project information
Repository
Merge requests 0
CI/CD
Security & Compliance
Deployments
Monitor
Infrastructure
Analytics
Settings
OU-VECHEV › rse-project
R
rse-project
Project ID: 8617
Leave project
Star 0
Fork 1
532 Commits
2 Branches
2 Tags
14.4 MB Files
940.4 MB Storage
master
rse-project /
History
Find file
Web IDE
Clone
added test guidelines
Benjamin Bichsel authored 2 days ago
5b86610b
Upload File
README
CI/CD configuration
Add LICENSE
Add CHANGELOG
Add CONTRIBUTING
Add Kubernetes cluster
Configure Integrations
Name
Last commit
Last update
analysis
deep pass on tests
2 days ago
docker
better dockerfile
1 year ago
.gitlab-ci-custom.yml
minor
1 year ago
.gitlab-ci.yml
2022 event project
1 week ago
README.md
minor improvements
1 week ago
README.md
RSE Project Development
In this file, we explain how to set up, run, and develop this project. Before reading this, you should read the project description.
Pipeline Status
the current status of the project.
Current pipeline status: pipeline passed
Current code coverage: coverage 78.06%
Pipeline status and test coverage

R rse-project
Project information
Repository
Merge requests 0
CI/CD
Security & Compliance
Deployments
Monitor
Infrastructure
Analytics
Settings

R rse-project

Project ID: 8617 Leave project

Star 0 Fork 1

532 Commits 2 Branches 2 Tags 14.4 MB Files 940.4 MB Storage

master rse-project / + History Find file Web IDE Clone

added test guidelines
Benjamin Bichsel authored 2 days ago 5b86610b

Upload File README CI/CD configuration Add LICENSE Add CHANGELOG Add CONTRIBUTING
Add Kubernetes cluster Configure Integrations

Name	Last commit	Last update
analysis	deep pass on tests	2 days ago
	better dockerfile	1 year ago
.gitlab-ci-custom.yml	minor	1 year ago
.gitlab-ci.yml	2022 event project	1 week ago
README.md	minor improvements	1 week ago

README.md

RSE Project Development

In this file, we explain how to set up, run, and develop this project. Before reading this, you should read the project description.

Pipeline Status

In the online version of this file, you can see the current status of the project.

Current pipeline status: pipeline passed

Current code coverage: coverage 78.06%

Frameworks

What to change

```
    @Override
    protected void merge(Unit succNode, NumericalStateWrapper w1, NumericalStateWrapper w2,
NumericalStateWrapper w3) {
        // merge the two states from w1 and w2 and store the result into w3
        logger.debug("in merge: " + succNode);

        // TODO: FILL THIS OUT

    }
```

What not to change!

```
package ch.ethz.rse.main;

import ch.ethz.rse.VerificationProperty;
import ch.ethz.rse.VerificationResult;
import ch.ethz.rse.VerificationTask;
import org.apache.commons.cli.*;

// DO NOT MODIFY THIS FILE

/**
 * Entry point for verifying a given program
 */
public class Main {

    public static void main(String[] args) throws ParseException {
```

Setup - Environment

We provide instructions exclusively for Linux (in particular, for Ubuntu). We strongly recommend you use Ubuntu to develop and run this project.

Docker

The project is set up to use docker - if you are curious, you can find a simple introduction here. Docker essentially simulates a lightweight virtual machine, which allows us to work in a controlled environment that comes with the necessary dependencies for this project. To install docker, you may follow these instructions.

We have prepared a docker image containing all tools needed to run and develop this project. To run this docker image in interactive mode (i.e., in a terminal where you can run any command), run

```
./run-docker.sh
[...]
root@a515c5af06d6:/project/analysis$ TYPE YOUR COMMAND HERE...
```

In the following, we assume you are inside the docker image, either because you ran the previous command, or because you are developing inside the container (discussed later).

Setup - Building

Maven

We manage the project's build, reporting and testing using Maven. The most important maven commands are:

```
# delete automatically generated files
root@a515c5af06d6:/project/analysis$ mvn clean
[...]
# compile, run unit and integration tests, report code coverage
root@a515c5af06d6:/project/analysis$ mvn verify
[...]
# generate test report
root@a515c5af06d6:/project/analysis$ mvn site
[...]
```

Command `mvn verify` generates information on test coverage here. Command `mvn site` generates a report on test results here.

Common issues

- If you get unexpected errors that do not make sense, it often helps to run `mvn clean` and try again.

Setup - Testing

Unit and Integration Testing

We have set up maven to run unit tests (detected by filename pattern `*Test.java`) and integration tests (detected by filename pattern `*IT.java`), which are located in this directory.

The most important maven commands regarding testing are

```
# run unit tests and create "surefire" report
root@a515c5af06d6:/project/analysis$ mvn test surefire-report:report
# run integration tests and create "failsafe" report
root@a515c5af06d6:/project/analysis$ mvn verify -Dskip.surefire.tests site -Dskip.surefire.tests
```

The two reports are located in surefire-report.html and failsafe-report.html, respectively.

Setup - Logging

Logging

We use SLF4J as a front-end for logging, and Logback as our logging backend.

While logging is not necessary to solve this project, it is easy to do and can be very helpful. See Runner.java for an simple usage example of logging in action.

Feel free to adapt the logging configuration if needed. For example, you can reduce the console logging information by adapting the "Log level for console".

Setup – CI & Coverage

GitLab CI/CD

We have set up the project to build and test the project on every push to the code repository, as controlled by .gitlab-ci.yml and .gitlab-ci-custom.yml. When the tests fail, you will be notified by e-mail (depending on your GitLab notification settings).

We recommend that you start to develop new functionality by writing corresponding tests, i.e., that you adhere to test-driven development.

Common issues

- `ERROR: Job failed (system failure)` : Likely, running the job again will resolve this issue. To this end, in GitLab, navigate to CI/CD -> Jobs -> Click on the box "Failed" of the failed job -> Click "Retry" (top right)

JaCoCo

We use JaCoCo to record and report the code coverage achieved by all tests. When running `mvn verify`, the code coverage is reported here. As discussed in the project description, we will award additional points for a instruction coverage of `>=75%`.

Important checks

Sanity Check for Submission

IMPORTANT NOTE: To ensure we will be able to run your submission, follow these rules:

1. Some files in this repository come with a note to `NOT MODIFY THIS FILE`. We will overwrite these files for grading, so changing them may mean that we cannot compile your project.
2. Before submission, check that the GitLab CI/CD runs without errors by checking the pipeline status (see above). In particular, please make sure that you do not commit failing tests as those could lead to errors in test coverage calculation. We will use the same workflow as the GitLab CI/CD to test your submission.

Setup - VSCode

Development (Optional but Recommended)

To develop, debug, and run this project, we suggest using Visual Studio Code to develop inside a docker container. This allows you to work on this project without having to install its dependencies on your host system.

Installation

To this end, follow these installation instructions. Instead of "adding your user to the docker group" (which opens up a potential security hole), you may instead install docker in rootless mode.

Then, install the Java Extension Pack in Visual Studio code:

- Open Visual Studio Code
- Launch VS Code Quick Open (Ctrl+P)
- Paste the following command, and press enter: `ext install vscjava.vscode-java-pack`
- Launch VS Code Quick Open again (Ctrl+P)
- Paste the following command, and press enter: `ext install ms-vscode-remote.remote-containers`

Project Description & Communication

- In your gitlab repository at

- Changes communicated via Moodle


Subscribe

- This presentation in your gitlab repository at

Analysis - Properties

Property NON_NEGATIVE:

- For any **reachable** invocation of `sell(v)` on an object `o` of class `Frog`, `v >= 0`.

Property ITEM_PROFIT:

- For any **reachable** invocation of `sell(v)` on an object `o` of class `Frog`, `o.production_cost <= v`.

Property OVERALL_PROFIT:

- Upon program termination, `Frog.total_profit >= 0`.
- If the program never terminates, this check is considered `SAFE`.

```
if (3 <= 1) {
    cool_frog.sell(-200);
}
```

Unreachable

We evaluate all properties independently!

➡ E.g.: OVERALL_PROFIT can hold even if ITEM_PROFIT does not

Analysis - Properties

Libraries

For your analysis, you will use the two libraries APRON and Soot. Part of your assignment is understanding these libraries sufficiently to leverage them for program analysis. In addition to the resources provided below, you may also consult

- The course lectures on abstract interpretation and pointer analysis.
- The language fragment of Soot to handle (see below).
- The documentation for APRON and Soot (including documentation of methods and classes), which is available in Visual Studio Code after setting up the project (see README.md file).

Apron examples
(extra resources should not be necessary)

APRON

APRON is a library for numerical abstract domains. An example file of using APRON exists here - it should demonstrate everything you need to know about APRON.

If you require more resources (which should not be necessary), you can also find documentation about the APRON framework here, and more extensive usage examples here.

Soot Documentation & Guides

Soot

Your program analyzer is built using Soot, a framework for analyzing Java programs. You can learn more about Soot by reading its tutorial, survivor guide, and javadoc. If you require more resources (which should not be necessary), you can find additional tutorials here.

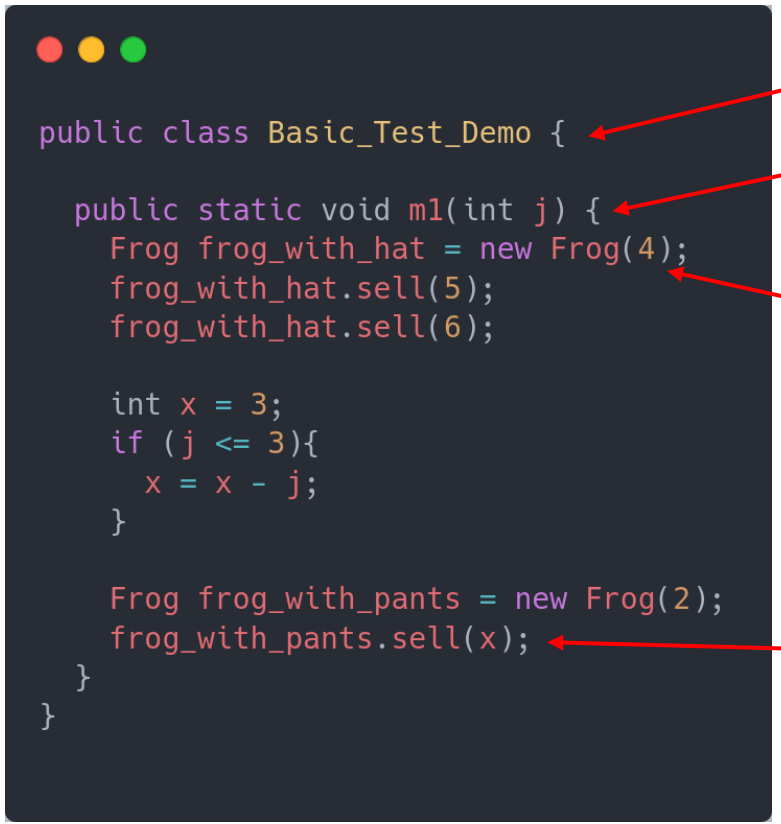
Your program analyzer will use Soot's pointer analysis to determine which variables may point to `Event` objects (see the Verifier.java file in your skeleton).

Analysis – Language Fragment

Jimple Construct	Meaning
DefinitionStmt	Definition Statement: here, you only need to handle integer assignments to a local variable. That is, `x = y`, or `x = 5` or `x = EXPR`, where `EXPR` is one of the three binary expressions below. That is, you need to be able to handle: `y = x + 5` or `y = x * z`.
JMulExpr	Multiplication
JSubExpr	Subtraction
JAddExpr	Addition
JIfStmt	Conditional Statement. You need to handle conditionals where the condition can be any of the binary boolean expressions below. These conditions can again only mention integer local variables or constants, for example: `if (x > y)` or `if (x <= 4)`, etc.
JEqExpr	==
JGeExpr	>=
JGtExpr	>
JLeExpr	<=
JLtExpr	<
JNeExpr	!=
IntConstant	Integer constant
JimpleLocal	Local variable
ParameterRef	Method parameter
JInvokeStmt	Call to `sell` (cp. JVirtualInvokeExpr) or initializer for new `Frog` object (cp. JSpecialInvokeExpr)
JReturnVoidStmt	Return from function
JGotoStmt	Goto statement

- Loops are also allowed in the programs.
- Assignments of pointers of type `Frog` are possible, e.g. `e = f` where `e` and `f` are of type `Frog`. However, those are handled by the pointer analysis.

Analysis – Add. Constraints



```
public class Basic_Test_Demo {

  public static void m1(int j) {
    Frog frog_with_hat = new Frog(4);
    frog_with_hat.sell(5);
    frog_with_hat.sell(6);

    int x = 3;
    if (j <= 3){
      x = x - j;
    }

    Frog frog_with_pants = new Frog(2);
    frog_with_pants.sell(x);
  }
}
```

Empty constructor

Single method in test class, **only integer parameters (no Frogs)**

Only integer constants (no local variables) in Frog constructor.

Argument: **Arbitrary part of the language fragment**

Analysis – Add. Tips

- You only need to track local variables for numerical analysis, use Soot pointer analysis for the heap
- You can assume the analyzed code never throws exceptions
- You can ignore overflows (assume that Apron captures Java semantics correctly)
- There may be loops: you will need to apply widening (but we ensure changing threshold doesn’t give extra points)
- APRON (Polka) is enough to evaluate all relations over locals
- We ordered the properties in increasing difficulty: NON_NEGATIVE, ITEM_PROFIT, OVERALL_PROFIT
- Don't crash, remain soundness by going to top

Deliverables

- The project deadline is **Wednesday, June 5th, 3, 17:00** (Zurich time)!
- We may decline to answer project questions after Monday, June 3rd, 2024, 17:00. This avoids last-minute revelations that cannot be incorporated by all groups.
- Commit and push your project to the master branch of your GitLab repository (that originally contained the skeleton) before the project deadline. **Please do not update your repository (commit, revert, etc.) after the deadline** - we will flag groups that try this.
- If you cannot access your GitLab repository, contact anouk.paradis@inf.ethz.ch.

Grading

- We will evaluate your tool **on our own set of programs** for which we know if they are valid or not.
- Your project must use the setup of the provided skeleton. In particular, you cannot use libraries other than those provided in analysis/pom.xml.
- We will evaluate you depending on the correctness of your program and the precision of your solution. You will not get points if your program does not satisfy the requirements. If your solution is unsound (i.e. says that an unsafe code is safe), or imprecise (i.e. says that a safe code is unsafe) we will penalize it.
- We will **penalize unsoundness much more** than imprecision.
- There will be a time limit of 10 seconds and 1GB of RAM to verify an application. Use this script if you want to ensure your solution adheres to these limits. Because the performance of a given solution is sometimes hard to predict, this limit is chosen generously.
- We will award **additional points** for groups that achieve a test instruction coverage of `>=75%` (achieving coverage `>75%` does not yield more points than achieving `75%`, and we will use the code coverage reported by JaCoCo in the README). If some of your tests fail, we will not award any additional points.
- Your solution must use abstract interpretation. Do not use other techniques like symbolic execution, testing, brute force, random guessing, machine learning, etc.
- Do not try to cheat (e.g., by reading the solutions from the test file)!
- Only submit solutions and test cases that **you have written yourself** and that you have understood. Cross-team implementation and copy paste (except from the project skeleton itself) is not permitted.

Grading

- We will evaluate your tool **on our own set of programs** for which we know if they are valid or not.
- Your project must use the setup of the provided skeleton. In particular, you cannot use libraries other than those provided in analysis/pom.xml.
- We will evaluate you depending on the correctness of your program and the precision of your solution. You will not get points if your program does not satisfy the requirements. If your solution is unsound (i.e. says that an unsafe code is safe), or imprecise (i.e. says that a safe code is unsafe) we will penalize it.
- We will **penalize unsoundness much more** than imprecision.
- There will be a time limit of 10 seconds and 1GB of RAM to verify an application. Use this script if you want to ensure your solution adheres to these limits. Because the performance of a given solution is sometimes hard to predict, this limit is chosen generously.
- We will award **additional points** for groups that achieve a test instruction coverage of `>=75%` (achieving coverage `>75%` does not yield more points than achieving `75%`, and we will use the code coverage reported by JaCoCo in the README). If some of your tests fail, we will not award any additional points.
- Your solution must use abstract interpretation. Do not use other techniques like symbolic execution, testing, brute force, random guessing, machine learning, etc.
- Do not try to cheat (e.g., by reading the solutions from the test file)!
- Only submit solutions and test cases that **you have written yourself** and that you have understood. Cross-team implementation and copy paste (except from the project skeleton itself) is not permitted.

Grading

- We will evaluate your tool **on our own set of programs** for which we know if they are valid or not.
- Your project must use the setup of the provided skeleton. In particular, you cannot use libraries other than those provided in analysis/pom.xml.
- We will evaluate you depending on the correctness of your program and the precision of your solution. You will not get points if your program does not satisfy the requirements. If your solution is unsound (i.e. says that an unsafe code is safe), or imprecise (i.e. says that a safe code is unsafe) we will penalize it.
- We will **penalize unsoundness much more** than imprecision.
- There will be a time limit of 10 seconds and 1GB of RAM to verify an application. Use this script if you want to ensure your solution adheres to these limits. Because the performance of a given solution is sometimes hard to predict, this limit is chosen generously.
- We will award **additional points** for groups that achieve a test instruction coverage of `>=75%` (achieving coverage `>75%` does not yield more points than achieving `75%`, and we will use the code coverage reported by JaCoCo in the README). If some of your tests fail, we will not award any additional points.
- Your solution must use abstract interpretation. Do not use other techniques like symbolic execution, testing, brute force, random guessing, machine learning, etc.
- Do not try to cheat (e.g., by reading the solutions from the test file)!
- Only submit solutions and test cases that **you have written yourself** and that you have understood. Cross-team implementation and copy paste (except from the project skeleton itself) is not permitted.

Grading

- We will evaluate your tool **on our own set of programs** for which we know if they are valid or not.
- Your project must use the setup of the provided skeleton. In particular, you cannot use libraries other than those provided in analysis/pom.xml.
- We will evaluate you depending on the correctness of your program and the precision of your solution. You will not get points if your program does not satisfy the requirements. If your solution is unsound (i.e. says that an unsafe code is safe), or imprecise (i.e. says that a safe code is unsafe) we will penalize it.
- We will **penalize unsoundness much more** than imprecision.
- There will be a time limit of 10 seconds and 1GB of RAM to verify an application. Use this script if you want to ensure your solution adheres to these limits. Because the performance of a given solution is sometimes hard to predict, this limit is chosen generously.
- We will award **additional points** for groups that achieve a test instruction coverage of `>=75%` (achieving coverage `>75%` does not yield more points than achieving `75%`, and we will use the code coverage reported by JaCoCo in the README). If some of your tests fail, we will not award any additional points.
- Your solution must use abstract interpretation. Do not use other techniques like symbolic execution, testing, brute force, random guessing, machine learning, etc.
- Do not try to cheat (e.g., by reading the solutions from the test file)!
- Only submit solutions and test cases that **you have written yourself** and that you have understood. Cross-team implementation and copy paste (except from the project skeleton itself) is not permitted.

Grading

- We will evaluate your tool **on our own set of programs** for which we know if they are valid or not.
- Your project must use the setup of the provided skeleton. In particular, you cannot use libraries other than those provided in analysis/pom.xml.
- We will evaluate you depending on the correctness of your program and the precision of your solution. You will not get points if your program does not satisfy the requirements. If your solution is unsound (i.e. says that an unsafe code is safe), or imprecise (i.e. says that a safe code is unsafe) we will penalize it.
- We will **penalize unsoundness much more** than imprecision.
- There will be a time limit of 10 seconds and 1GB of RAM to verify an application. Use this script if you want to ensure your solution adheres to these limits. Because the performance of a given solution is sometimes hard to predict, this limit is chosen generously.
- We will award **additional points** for groups that achieve a test instruction coverage of `>=75%` (achieving coverage `>75%` does not yield more points than achieving `75%`, and we will use the code coverage reported by JaCoCo in the README). If some of your tests fail, we will not award any additional points.
- Your solution must use abstract interpretation. Do not use other techniques like symbolic execution, testing, brute force, random guessing, machine learning, etc.
- Do not try to cheat (e.g., by reading the solutions from the test file)!
- Only submit solutions and test cases that **you have written yourself** and that you have understood. Cross-team implementation and copy paste (except from the project skeleton itself) is not permitted.

Grading

- We will evaluate your tool **on our own set of programs** for which we know if they are valid or not.
- Your project must use the setup of the provided skeleton. In particular, you cannot use libraries other than those provided in analysis/pom.xml.
- We will evaluate you depending on the correctness of your program and the precision of your solution. You will not get points if your program does not satisfy the requirements. If your solution is unsound (i.e. says that an unsafe code is safe), or imprecise (i.e. says that a safe code is unsafe) we will penalize it.
- We will **penalize unsoundness much more** than imprecision.
- There will be a time limit of 10 seconds and 1GB of RAM to verify an application. Use this script if you want to ensure your solution adheres to these limits. Because the performance of a given solution is sometimes hard to predict, this limit is chosen generously.
- We will award **additional points** for groups that achieve a test instruction coverage of `>=75%` (achieving coverage `>75%` does not yield more points than achieving `75%`, and we will use the code coverage reported by JaCoCo in the README). If some of your tests fail, we will not award any additional points.
- Your solution must use abstract interpretation. Do not use other techniques like symbolic execution, testing, brute force, random guessing, machine learning, etc.
- Do not try to cheat (e.g., by reading the solutions from the test file)!
- Only submit solutions and test cases that **you have written yourself** and that you have understood. Cross-team implementation and copy paste (except from the project skeleton itself) is not permitted.

Web interface to master solution

We provide a web interface so you can query the master solution!

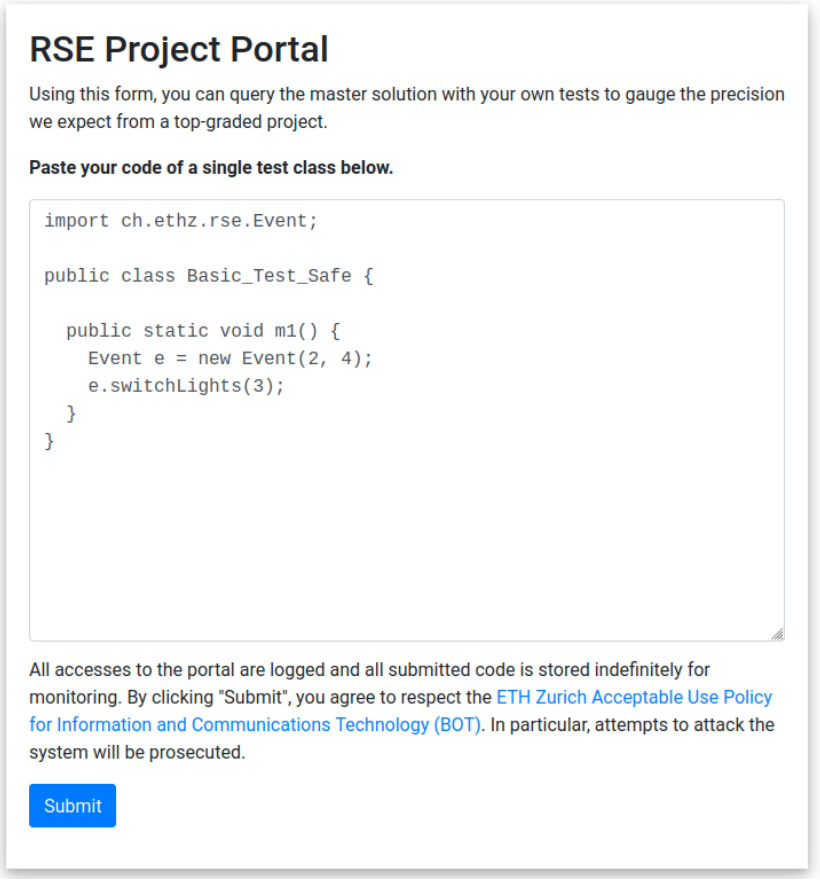
https://rseproject.ethz.ch/rse-project

Hall of Fame

These students have found unsound behavior in the master solution:

- (no unsoundness was found so far)

Contact anouk.paradis@inf.ethz.ch if you have found unsound behavior.



Getting help

For questions about the project, please consult (in this order):

- This project description
- The skeleton in your GitLab repository (in particular the README file)
- The documentation of libraries&frameworks, in particular APRON and Soot discussed above
- The Moodle page. All students will see and are encouraged to reply to the questions about the project.
- The project TA at anouk.paradis@inf.ethz.ch (only when Moodle is not possible)

Last Tips

- Don't get overwhelmed and ignore the project
- Start with SOUNDNESS then try for more PRECISION

  (points for partial sound solutions)
- Don’t ignore the project: 20% of the grade means you need a 5 at the exam to pass without it
- Nest steps:
  - Read project.md
  - Read readme.md
  - Look for FILL THIS OUT in files
  - Tackle properties in order of difficulty and test often

Soundness/precision

Sound analysis

if SAFE then for all executions assertions are verified.

if UNSAFE, we don't know

Precise analysis

reduce "false" UNSAFE