SYLLABUS FOR TEACHING PROGRAMMING FOLLOWING LIGHT OOP PRINCIPLES

Co-funded by the
Erasmus+ Programme
of the European Union

Project	Object Oriented Programming for Fun
Project acronym	OOP4FUN
Agreement number	2021-1-SK01-KA220-SCH-00027903
Project coordinator	Žilinska univerzita v Žiline (Slovakia)
Project partners	Sveučilište u Zagrebu (Croatia) Srednja škola Ivanec (Croatia) Univerzita Pardubice (Czech Republic) Gymnazium Pardubice (Czech Republic) Obchodna akademia Povazska Bystrica (Slovakia) Hochschule fuer Technik und Wirtschaft Dresden (Germany) Gymnasium Dresden-Plauen (Germany) Univerzitet u Beogradu (Serbia) Gimnazija Ivanjica (Serbia)
Year of publication	2023

Table of contents

List of figures

List of tables

1. Information sheet

1.1. Subject annotation

The aim of the course is to teach students to solve programming tasks using basics of object-oriented programming (OOP) following light OOP paradigm. Students will learn to split given tasks among cooperating objects; to determine their competencies; and to implement designed model. The course does not require previous programming skills. It is taught in Java programming language. The course explains light OOP concepts (such as encapsulation, inheritance, or association) on the creation of computer games, where these concepts are simply and intuitively utilized. The process of creating a computer game is based on teamwork and practically utilizes knowledge and skills from other areas of informatics and to it related subjects (work with multimedia and office software). The design of every computer game is open enough for students to expand the game individually and creatively. Moreover, the design leads to the proper utilization of acquired knowledge.

1.2. Subject characteristics

Subject is focused to introduce innovative approach to teach programming, based on the solving of tasks using the object-oriented programming (OOP) paradigm. OOP is nowadays the dominant paradigm for application development. Therefore, it is proper for students to possess the knowledge and skills in this area. The subject presents development environment that utilizes different forms of source code editing (frame-based editing using simplified form as well as real source code writing) what makes it possible to teach students on different levels of prior technical knowledge and activity. With its simplicity and clarity this tool supports quick and intuitive comprehension of taught topics what has positive influence on students and their motivation.

1.3. Aim of the subject

Via programming of interactive games in graphical environment, the student will gain knowledge and skills, so that student will be able to:

- identify a problem,
- identify suitable objects to solve identified problem (object decomposition),
- design classes of objects, as well as their attributes and methods,
- identify and properly utilize objects relationships (association, inheritance),
- design an algorithm to solve problem and distribute it among cooperative objects,
- use source code elements (branching, loops) to implement designed algorithm,
- effectively use means for source code debugging,
- create simple application with graphical interface in the Greenfoot environment.

1.4. Learning outcomes

Learning outcomes of the subject are summarized as follows:

- understanding the basic principles of object-oriented programming,
- understanding the basics of algorithmization,
- understanding the syntax of the Java programming language,
- analyzing program execution based on the source code,
- the ability of creating own programs with the use of OOP.

1.5. Material and technical requirements

PC classroom containing separate workspace for every student plus workspace for teacher. Workspace is considered as table, chair, personal computer (PC). All workspaces should be connected to LAN network with access to internet (recommended).

PC should meet following minimal requirements:

- operating system (either Microsoft Windows 7 or later, Linux (Debian), Mac OS 10.10 or later),
- office software with text, table and presentation editor (e.g. Microsoft Office, Libre Office, Open Office),
- java SE Development Kit (JDK),
- Greenfoot environment (version 3.8 or later),
- simple graphical software,
- web browser (e.g. Edge, Google Chrome, Mozilla Firefox, Opera),
- relevant software for other PC hardware.

2. Syllabus principles

Proposed syllabus is designed to tackle problems identified in PR1 and PR2 (see chapter "Aligning results with PR1 results" in PR2 report). In the following table we present perspective on curriculum development, learning outcomes, teaching materials and teaching activities as proposed in PR2. Following these findings, we were able to formulate syllabus principles.

PR2 findings	PR3 syllabus principles
In high schools, OOP should be introduced by topics covering basic programming concepts in the beginning and narrower topics related to OOP would be more appropriate in separate courses. It is crucial to connect and encourage information exchange between school and university teachers, with the involvement of policymakers who define curricula related to programming skills at all educational levels. From a course/instructor perspective and from a course design perspective, the following innovative forms of instruction/knowledge transfer should be used: Blended learning, learning-by-doing, problem solving, collaborative problem, teamwork, problem-based learning, active learning, lab-based learning. In addition, various forms of innovative approaches should be applied in lectures, seminars, and laboratory exercises. Games and gamification in general were frequently used to motivate students to program. Students liked the opportunity to be creative or to compete for knowledge when supported by an appropriate setting. From the results of the literature review, three different types of learning through games were suggested: learning by playing, learning by creating games, learning by using game-related tools and learning with gamification.	Syllabus must properly utilize OOP from the very beginning following object first approach. The suitable level of OOP for high schools has been identified and formulated as light OOP. Light OOP can be with benefits used when creating games, since it is easy to identify the objects of the game as well as objects' competencies and properties. To motivate students using games it is important to make students interested in the games. There should be created several game-based projects with different mechanics to fulfill this goal. Moreover, several games that will be prepared will allow to: • create different teaching materials targeting different teaching conditions (online/onsite, intensive/whole year course, beginner/advanced students). This is a key property for upcoming project results, • build one game with the teacher present and other games as home assignments (teacher will be able to see if students can apply knowledge in different context), • build a game according to given instructions with minimal teacher's interaction, so that students will understand importance of well written technical description and they will apply learned skills to create game as specified, • introduce a new light OOP concept as game introduces new mechanics. This will make it possible to stop development of the project if the teacher decides to cover subset of light OOP because of nation specifics.
For learning and teaching of OOP concepts, learning by creating the games showed the significant effects for improving students problem solving skills and engaging them in a fun and entertaining environment.	
As several PR2 outputs point out, the main goal should be to incorporate learning and teaching tasks into stimulating and entertaining activities	Projects will be built using learning by doing principle. We try to minimize the theory

that will have a positive impact on higher attendance and completion rates. This would increase the interest of high school students for programming in general and eventually lead to better understanding of programming and OOP concepts. In that case students would not be "lost" when faced with university curricula.	explanation and support investigation, practice, and production aspects of learning. Working in groups encourages students that may struggle in the beginning. If the project is developed in a group, agile methods of development as well as teaching can be used. Before implementation, the teacher may decide to use analytical and design phase of the project. This will make it possible to utilize knowledge and skills of students gained in previous lectures/courses as well as to engage students in project development. Students may be asked to: • work with information sources to find proper game, • formulate game rules and/or requirements on their application (either in the oral or written form), • prepare multimedia (images/sounds).
The overall goal of PR2 was to find appropriate and innovative learning and teaching ideas and approaches that would solve these issues. As mentioned in the previous chapters[1] there are several identified good practices that could be used to improve the achievement of learning outcomes during the high school education. However, it should be noted as well that curriculum redesign should result in introduction of OOP topics and set goals in achieving OOP related learning outcomes as well. It is also important to select an appropriate type of assessment: (online) questionnaires are the only accepted method for assessing students' enjoyment, usefulness, interest, engagement, and simplification of programming and OOP concepts which will be defined on the high-school and not on university level.	With focus on learning by doing principle we tried to minimize acquisition type of activities and to strengthen the investigation part. Projects will be built in a way that enables simple expansion, so motivated students will be capable of working on the projects by themselves. To validate the proposed curriculum, we will prepare learning design for every project. Then we will compare analytical output of new projects utilizing light OOP with analytical output of learning design built for project that was developed in past and is already deployed in praxis in Slovakia (among other schools in country, it is used by project partner Obchodna akademia Povazska Bystrica) and Czech Republic (used by project partner Gymnazium Pardubice). Positive feedback for this validated project has been published in PR2, therefore we assume that following the same best principles and practices, we will transfer positive acceptance of proposed curriculum.
By using teamwork in OOP assignments, students would have the opportunity to share their knowledge and transfer the implementation of basic programming concepts to other students (peer-to-peer learning).	When designing here proposed game-based projects, we kept in mind utilization of different techniques, such as EduScrum. For this reason, we deliver each project in the form of GIT repository and supply proper knowledge to teachers. While it is not limited to utilize this approach and teachers may still use traditional
This project's results will yield a set of materials which will give a chance to highly motivated	

[1] See previous chapters of PR2 report

students with solid prior-knowledge to increase that knowledge through different activities and roles. Such students could significantly improve the overall achievements of the whole group if they will be given a chance to share the knowledge or to lead the teams.	approach, using agile techniques will lead to teach:
	• Basics of versioning systems - we suggest GIT as the most utilized versioning system, using versioning system makes it easy to
	  ○ share source codes and
	  ○ develop new features of the games, without impact on the flow of the project (e.g. in a separate branch), what will motivate ambitious students.
	• Teamwork – each student will be responsible for the specific aspect of a game leading to the necessity of effective communication between team members.
	• Time management – individual parts of the projects will have to be delivered on time in order to merge them and continue with other work, however, proper use of versioning system and OOP open many possibilities to deal with time struggles what may lead to positive motivation of students even during hard times.
	
	We will realize multiplier events with teachers covering both introduction to GIT as well as the utilization of principles of agile software development in teaching. Students will form project teams, with specific roles, they will share ideas on stand-ups, give and assign themselves goals, they will realize sprints, create documentation and other artefacts and they will present their solution.

Use of more different tools and programming languages in earlier years of study should be encouraged. Programming concepts could be simplified using visualization tools. Although some countries have already introduced the use of some tools like Logo or Scratch, these are interesting to elementary schools but not to high schools. Thus, more advanced tools that are designed to support OOP should be used. We have recognized that Alice and Greenfoot stand out among other tools.	We use Greenfoot environment that utilizes Java programming language. Java is currently very popular and in praxis widely used programming language. Moreover, the Greenfoot presents the frame-based source code editor using Stride language. This opens possibilities for teachers who will want to use in this syllabus presented techniques with students of younger age. Greenfoot is very visual and from the beginning it makes it possible to create a visualized object, that is “alive” and can be interacted with. Therefore, the theoretical introduction is

	minimized, and students will start working from the very beginning.
As students reported, for teaching programming it is important that teachers use novel and up-to-date teaching materials and employ creative teaching methods. Also, the teacher's availability and flexibility to work with students outside the classroom is necessary to motivate students and generate greater interest in the subject.	Using presented principles, we create modern syllabus for teachers that will cover light OOP topics and is based on project work and genuinely utilizes object first approach. We propose several game-based projects, each delivered in the form of GIT repository. For every project there is created learning design what made it possible to validate these projects with already in praxis utilized and positively admitted approach.

The content and scope of the educational program differ with regards to the game project developed during classes. For detailed analysis refer to respective learning design and to attached files of analysis.

3. Projects

With regards to the syllabus principles there have been created two game projects that properly utilize object-first and learning by doing principles. Moreover, we processed project Bomberman, that was a backbone project of national project IT Academy, that is used in praxis in Slovakia. We use this project for validation of proposed projects. Organization of all project chapters is as follows.

- **Project description** – the basic description of the game with screenshot of the finished application and summarized game rules. Project description also presents the connection to the light OOP topics.
- **Link to source codes** – source codes are organized in the form of GIT [1] repository. Using properly managed repository introduces the teacher to use the modern approach of source code management. We have used GIT as a versioning system that is among the most popular in recent years [2]. GIT also enables us to use cloud-based repositories, such as GitHub [3] or GitLab [4], that are free and offer to use many tools to enhance team collaboration. Every repository is organized as follows:
    - **Branch per topic** – tasks of every topic from the syllabus are developed in dedicated branch. Master branch contains only initialization commit and merges of topic branches.
    - **Commit per task** – every task that is oriented to produce/modify source code is in the form of commit. Description of commit matches respective task's number.
- **Link to learning design** – syllabus is built in the form of learning design. Learning design enables us to define learning outcomes (that cover necessary competencies identified in PR1 and PR2) and associate them with topics (see the connection to the branches of respective GIT repository). The topics are organized into units that are composed of the TLAs (see connection to the commits in respective GIT repository). Using learning design makes it possible to analyze the time allocation what is directly connected to the validation of declared learning-by-doing principle. Since the learning design was processed also for already established and in pedagogical praxis used project (Bomberman), it makes it possible to identify the potential problems in design. To be able to perform validation, the projects are organized into topics in the same way.
- **Covered topics of light OOP**.
- **Content and scope of the educational program** – overview of learner workload in specific learning type as well as overview of contribution of topics to individual learning outcomes.
- **List of topics**. Every topic contains:
    - brief description,
    - comparison of learning designs,
    - list of tasks.

3.1. Bomberman

Bomberman is a fairly well-known multiplayer game. The game takes place in an arena that contains the players as well as some fixed obstacles. The player can plant bombs that explode after a certain amount of time. The aim of the game is to eliminate the opponents by using bombs. Some of the obstacles in the arena can be destroyed using bombs. After destroying an obstacle, a random bonus can appear in the game, which for example increases the speed or power of the player's bombs. The game ends if there is only one player left in the game, in which case that player wins, or if there are no players left in the game, in which case the game ends in a draw.

The Bomberman project covers most of the topics of light OOP. It focuses on the main aspects of the OOP which are summarized in the topics below. Furthermore, it introduces some topics reaching beyond light OOP such as generating and using random values using the **`Random`** class.

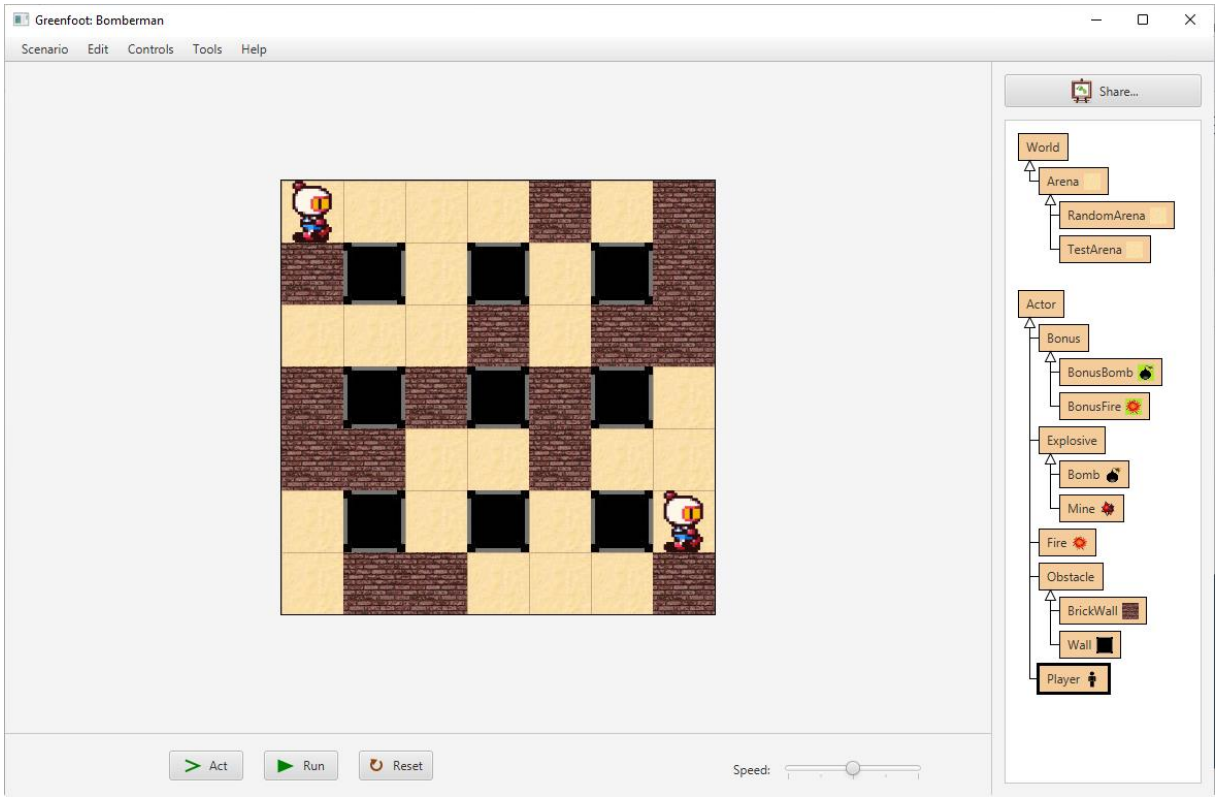


Figure 1: Greenfoot environment with the final state of project Bomberman

Source codes are available at:
https://gitlab.kicon.fri.uniza.sk/oop4fun/project-bomberman

Bomberman project is specific in a way that it contains commits that include work that is to be done in between two consecutive tasks. Such commits are described using messages of the form “Code X.Y”, where X is the number of the topic and Y is the number of intermediate task within given topic. In the text, the commits are labeled as “Intermediate commit”.

Learning design is available at:
https://learning-design.eu/en/preview/70bcf65d805b0603f6c1aeab/details

3.1.1. Content and scope of the educational program

Overall learner workload is 49h 30min and is distributed as follows:

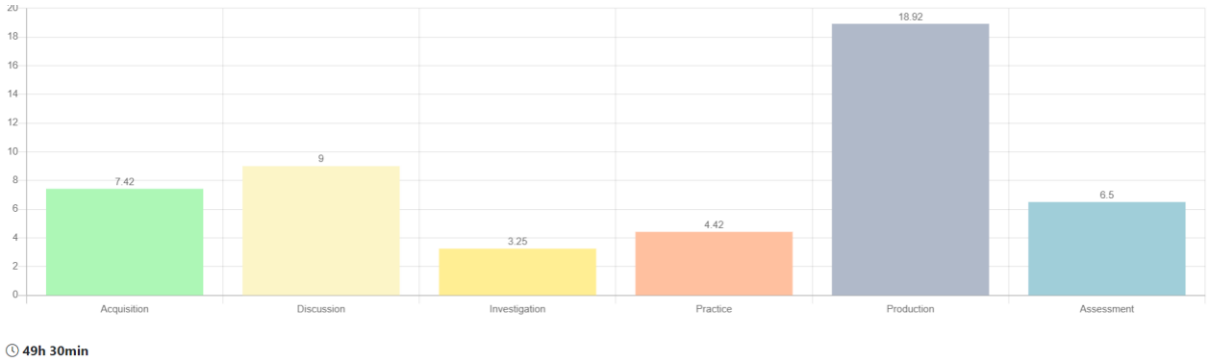


Figure 2: Learner workload when using project Bomberman

Constructive alignment is summarized in table below:

Table 1: Constructive alignment of project Bomberman

Topic	Assessment		Understanding the basic principles of object-orien... (25)	The ability of creating own programs with the use ... (20)	Understanding the syntax of the Java programming l... (10)	Understanding the basics of algorithmisation (25)	Analysing program execution based on the source co... (20)
	Formative	Summative					
Greenfoot environment	0	0		80%	20%		
Class definition	0	35	60%	20%	20%		
Algorithm	0	20		10%	10%	60%	20%
Branching	0	20		10%	10%	70%	10%
Variables and expressions	0	5		10%	10%	70%	10%
Association	0	10	60%	10%	10%	10%	10%
Inheritance	0	0	50%	30%	10%		10%
Loops	0	40		40%	10%	40%	10%
Lists	0	0		50%	10%	30%	10%
Encapsulation	0	15	50%	30%	10%		10%
Polymorphism	0	15	50%	20%	10%	10%	10%
Random numbers	0	20		30%	10%	50%	10%
Total	**0**	**180**	**270%**	**340%**	**140%**	**340%**	**110%**
	180						

For a detailed plan refer to attachment 5.1.

3.1.2. Topics

Project Bomberman is divided into ten topics:

Covered topics of light OOP are:

- classes, objects, instance
- methods, passing methods arguments
- constructors
- attributes
- encapsulation
- inheritance
- abstract classes
- object live cycle

1. Introduction to Greenfoot environment

The topic is devoted to basic project setup. Students will learn how to set the dimensions and appearance of the environment, create a class (as a subclass of the **`Actor`** class), create its instance, send it a message, and observe its internal state.

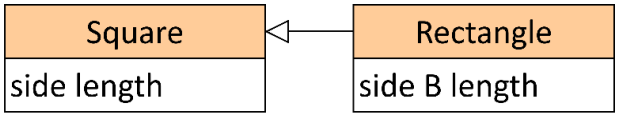
Commit: 84de89768134d119dbe94017fe477152c4307b61

1.1. Identify objects

Identify objects in your surroundings and list their properties and actions that they can perform. Can you identify objects that have no properties? Can you identify objects that can do nothing? Can you identify immaterial objects (ones that we can't physically touch)?

1.2. Validate object hierarchy

The following figure shows the hierarchy of the Square and Rectangle classes. Is this a good hierarchy?



1.3. Create a simple hierarchy

Create a class hierarchy of:

a) means of transport,
b) animals,
c) and computer parts.

List the properties that each class defines. List the properties defined by each class. Which classes are non-material in the design, so we cannot touch them? We will call such classes abstract in the future.

1.4. Create a tile

In a graphical editor, create a tile that will graphically represent a cell of the world. Select square representation, ideally 60x60 pixels. Import or save the image in the project folder in the **`images`** folder. Set the image as the world image. This is done by right-clicking on **`MyWorld`** and selecting `Set image...` . Note that the **`MyWorld`** class has been given a small icon in the class diagram to represent its graphical form.

Commit*:* 5eeecea8bbff323caab71d0a25068d596891d447

1.5. Create the world

Modify the **`MyWorld`** class constructor to create a 25x15 cell world, with each cell being 60 pixels in size. How would the image need to be modified to make the world look like a chessboard (i.e., with alternating different colored squares)?

Commit: 86da438686edad7900e8e11f595917f49209d346

Next, create a **Player** class. This is done by right clicking on the **Actor** class and selecting the **New subclass...**. Name the subclass **Player** and select its image from the library. Again, notice the small icon next to the **Player** class similar to the **MyWorld** class. To create an instance of the **Player** class, right-click on the **Player** class, select **new Player**, and move the icon with the player to the background of the world. Left-click to place the instance. To view the status of an instance, right-click it and select **Inspect**.

Intermediate commit: 316e48b29455649483c53080c6f5d49d74cf09c3

1.6. Inspect player state

Grab the created instance of the **Player** class with the mouse and move it to another position in the world. Watch a live view of the internal state – what do you see? Create another instance of the **Player** class and view its internal state as well. Again, drag one of the two instances with the mouse – which internal state has changed?

1.7. Interact with players

Call the methods provided by the Greenfoot environment over different instances of the **Player** class. To do this, left-click on the instance and select, for example, the method **void move(int)**. When prompted, enter an integer. Observe how the internal state of the instance changes.

2. Algorithm, application controls, method creation

This topic covers the creation of public methods that move the player in the world. It also introduces Greenfoot environment tools that control the execution of the scenario.

2.1. Write a simple algorithm

Write down the procedure, how to prepare coffee, how to travel to school, and how to cook lunch.

2.2. Write a more general algorithm

Make a general algorithm for the preparation of a hot drink. Think about what the inputs of such an algorithm need to be for it to be general.

2.3. Inspect a class instance

Explore the methods of the **Player** class instance. To do this, right-click on the class and select **Open Editor**. What do you observe? By analogy to the **act()** method, add the **makeLongStep()** method.

Intermediate commit: 13b7cca1040b3e1783a8819552d904a05f732817

2.4. Implement a method

Add a statement to the body of the **makeLongStep()** method such that the instance of the **Player** class is moved by two cells in the current direction. Then create multiple instances of the **Player** class and invoke this method on each instance. Is the behavior expected?

Commit: e678e104ec9b3bac0cc52a11f0a897548cdb5e82

2.5. Add documentation

Add a documentation comment for the **makeLongStep()** method.

Commit: 5c04e53c250331bdbf98a01d0c97c722486f8c79

2.6. Add more documentation

Edit the documentation comment of the **Player** class. Add the version of the class and its author.

Commit: 943aedd847dd93796f5a63af824e0556f45b208f

2.7. Read the documentation

Explore the documentation window.

2.8. Add player action

Modify the body of the **`act()`** method of the **`Player`** class to call the **`makeLongStep()`** method.

Commit: 24fca7f90c940830674d9ee51c28988119a18d83

2.9. Explore application controls

Try the buttons that control the app. Create multiple instances of the **`Player`** class. Press the **`Act`** button – what happens? Press the **`Run`** button – what happens? After the first press of the **`Run`** button press the **`Pause`** button, what happens? What effect does the **`Speed`** slider have on the **`act()`** method call after the **`Run`** button is pressed? What happens, when you press the **`Reset`** button?

2.10. Add another player action

Add a method to the **`Player`** class that will walk an instance of the **`Player`** class in a square. Document your method. Use appropriate methods from the **`Actor`** base class to move and rotate. Modify the **`act()`** method so that an instance of the **`Actor`** class walks in a square when it is called. Then verify your solution by running your application.

Commit: 4721b0e7cceaa883d242d0011485cb9a0ca1d742

3. Branching and player control

This topic introduces students to branching in the form of **`if-else`** statement and **`switch`** statement. The branching is used in the detection of the edges of the world and collisions with walls – which are new objects added to this topic.

The following intermediate commit shows a modification to the **`act()`** method to make an instance of the **`Player`** class move one field and rotate when it contacts the world edge.

Intermediate commit: cd358b49452e8fb3342aba073651fa969a56b275

3.1. Move the player

Modify the code of the **`act()`** method in the **`Player`** class so that the player only moves when the **`M`** key is pressed (**`M`** as a move). Keep the code responsible for turning the player when it reaches the edge of the world but think about its location. When can a player turn be performed?

Commit: 9d254e0bf2a4c49bd82fd15858d7a3104ab201c7

3.2. Observe the player's state

Create an instance of the **`Player`** class and place it in the center of the board. Open a window with the internal state of the instance and position it so that it is visible while the application is running. Then run the application and observe how the values of the **`x`** and **`y`** attributes in the **`Player`** class change. How do these values change as you move up, down, left, and right? Use different values for the **`setRotation()`** method (0, 90, 180,270) try to replace the **`isAtEdge()`** method with the **`getX()`** and **`getY()`** methods.

Intermediate commit: f28065cf775055bdd9fc41757af31b818fb76552

3.3. Add world edge detection

Add code to the body of the **`act()`** method to properly rotate the player after reaching the bottom and left edges of the world.

Commit: 1dba08bd12adbf8288087c1957192f8d58745b6e

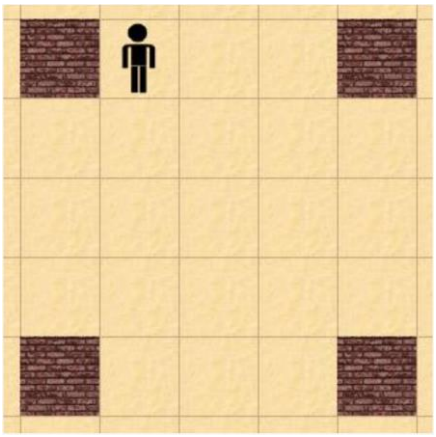
3.4. Add walls

Create two new classes as subclasses of the **`Actor`** class. The first class will be **`BrickWall`** class, and the second class will be **`Wall`**. Prepare suitable 60x60 pixel images in a graphical editor. Then assign these images to the newly created classes.

Commit: ce0c50360a562e2a58592ca68b8a2ac83aa98c10

3.5. Observe the player's movement

Create four instances of the **`BrickWall`** class and one instance of the **`Player`** class as shown in the figure below. Guess how the player will move? Run the application. Does your prediction match what you observe?



3.6. Add wall collision detection

Add code to the **`act()`** method of the **`Player`** class to ensure that the player turns 90° counterclockwise when he enters a cell that contains an instance of the **`Wall`** class.

Commit: 8dd56c9f4ebb1ece0037eff1a4f8a1efc6425048

In the following intermediate commit, the behavior of the **`Player`** class is modified. Full branching is used and additional conditions are added. Consider how an instance of the **`Player`** class behaves if it enters a cell that contains an instance of the **`Wall`** class.

Intermediate commit: 61415cb24c76a38280e3d82a9a5cf5104d72c5e9

3.7. Predict player movement

Predict how an instance of the **`Player`** class will move. Does the result agree with your prediction?

3.8. Observe edge detection

Place one instance of the **`Player`** class in the corners of the world. Predict how this instance will move when the application is started. Does the result agree with your prediction?

3.9. Finish wall collision handling

Complete the cascade of conditions so that touching an instance of the **`BrickWall`** class and the **`Wall`** class is checked only if the player is not on the edges of the world. Check the instance of the **`BrickWall`** class first.

Commit: 6f1af5aecda75e6b9a0198b3ee01f22cca4ad182

3.10. Move the player automatically

Create a method to automatically move an instance of the **`Player`** class. Move all code from the **`act()`** method into the new method. The method identifier can be, for example, **`moveAutomatically`**.

Commit: 52c3706bf61f43e9db82835a5d09274fdff1efc7

3.11. Move the player using arrows

Create a **moveUsingArrows()** method in the **Player** class. Program this method so that the player only moves when an arrow is pressed. It will move in the direction of the pressed arrow. Take care to keep the code efficient. In the **act()** method, call the new method.

Commit: 97e02be7423c9da1ea466547c944b9d58a26fa60

3.12. Prepare images

Prepare four images for the player to move up, down, left, and right. The dimensions of the images must not exceed the dimensions of the cell, in our case 60x60 pixels. Place the prepared images in the **images** directory.

Commit: 08df8b3d3224ddead2c54859b3734e5de6455acf

3.13. Use the images

Create an **updateImage()** method that changes the player's image according to his current rotation. Add a call to this method to the body of the **act()** method. The following commit shows the possibility of using **switch** statements in branching, as an alternative to **if-else**.

Commit: c564ef0bf67120d3b846b0014a5dd1e24c779dec

3.14. Run the application

Start the app. Notice that the images are adjusting, but they rotate according to how the player is rotated. Solve this problem.

Commit: 81f3be530f0acd454bbe1a0215357628086d547c

3.15. Use more players

Try creating multiple players and controlling them using the keyboard. What do you observe?

4. Variables, expressions, and advanced player control

This topic deals with the introduction of variables in the form of class attributes and method parameters. Students will use the attributes to control the speed of the player as well as setting specific keys that control the movement of a player. This will allow the game to have multiple players each controlled by different keys.

4.1. Spot the difference

What is the difference between the following algorithms?

1.
```
int a;
boolean c;
...
if(a > 0){c = true;}
if(a < 0){c = false;}
```

2.
```
int a;
boolean c;
...
if(a > 0){c = true;}
else {c = false;}
```

4.2. Write simple expressions

Write an expression using Boolean and relational operators to express that a variable of type **int** a has a value belonging to the following intervals: **<-10,10), (5,142), (-11,-3) OR (1,25>**.

4.3. Evaluate expressions

Choose the values of the variables **x** and **y** and add them to the expressions. What will be the values of the variables **b1** and **b2** in each case (1 and 2)?

1.
```
int x, y;
...
boolean b1 = x > 0 && y == 1;
boolean b2 = x <= 0 || y <= 0;
```

2.
```
int x, y;
...
boolean b1 = (x > 0) && (y == 1);
boolean b2 = (x <= 0) || (y <= 0);
```

Intermediate commit: 775935e5d09d4db8b5b41ea27d35b0595cfbdb3b

The previous intermediate commit modifies the **Player** class. It adds **String** attributes **upKey**, **downKey**, **rightKey** and **leftKey**. These will be the keys that will be used to control the movement of an instance of the **Player** class. The keyword **private** means that the attributes will only be available within the **Player** class. Within this class, the attributes will be accessed via the word this (i.e., an instance of this class), i.e., **this.upKey**, **this.downKey**, **this.rightKey**, and **this.leftKey** Further, it modifies the **moveUsingArrows()** method by causing the **"left"** key to be replaced by the **this.leftKey** attribute. The other keys are similarly replaced. The keys to control the movement of a given instance must be specified when the instance is created. To do this, a constructor must be created. This is a special method for creating an instance of a given class, which is executed when it is created. The constructor again has four parameters - **upKey**, **downKey**, **rightKey** and **leftKey**. Inside the constructor, you need to distinguish between **leftKey** and **this.leftKey**. The former is a parameter of the constructor and the latter is an attribute of the instance of the class. The constructor is not mandatory within the class. If we don't implement it, the default with empty code will be used.

4.4. Test the constructor

Test the constructor and the modified method for controlling player movement by inserting two instances of the **Player** class into the world. For each instance in the dialog, set different keys to control its movement. Test to see if the inserted players can be controlled independently.

4.5. Rename a class

Rename the **MyWorld** class to the **Arena** class. Note that the constructor name must be the same as the class name. Test how the Greenfoot environment behaves if it is not.

Commit: 7f85bea606e71344e376ab7519be3082fec0ee7c

Until now, it was always necessary to manually add instances of the class, which disappeared after clicking the **Reset** button. This is because there are no instances created in the constructor of the **Arena** class. In the following intermediate commit, the **Arena** class attribute **player1** of type **Player** is created. By being **private**, it is only available within the **Arena** class. It is then created using new and placed at the specified position using the **addObject** method. Let's add that the keyword **new** triggers the constructor of the class.

Intermediate commit: d816af6e4f043848d4ef16ac96c80134354d974b

4.6. Add one more player

Add another player to the world, for example by using a **`player2`** reference attribute, which will be controlled by the **`w`** – up, **`s`** – down, **`d`** – right, and **`a`** – left keys. Place the player at coordinates **`[24,14]`**. Once added, test the controls.

Commit: b69a02f684f702ef963c6adfa08bd8eecc7fd61f

4.7. Reference attributes

What would happen if we made the assignment **`this.player1 = this.player2;`** after creating both players? Would that be a problem?

4.8. Extend the player class

Extend the **`Player`** class with an additional attribute of type **`int`** representing the player's step size. Here two constructors are used, differing in the number of parameters. This is the so-called overloaded constructor. The one with the same parameters is always used.

Commit: 6b4c1680a7ea0b4a690d1671869436ca8e5a27ce

4.9. Integrate step size

Modify the **`moveUsingKeys()`** method to respect the new attribute step size. Create a new instance of the **`Player`** class and test the functionality of the program.

Commit: f6bc391828f6b3a08a58d28a9d89b5a7d75eb003

4.10. Make players move at different speeds

Your task is to ensure that the player moves one cell at a time but at different speeds. Each player can have a different speed. As a hint, you can program different movement speeds by, for example, not moving the player every time a key press is detected (we do the detection in the **`act()`** method, so if you hold down a key, you will detect its press every time it is executed), but only on every N-th time it is executed. The larger the N, the lower the player's speed will be. How do you enter N? Where do you store it? How will you know how many keystrokes have elapsed? (Hint: a counter is also needed).

Commit: 395bc78a6d243dcd1f471c385e6d010718cc6e44

5. Object and class cooperation

The main focus of this topic is object collaboration. In this topic, students will add interaction between objects in the arena to the project – for example, making sure that the player cannot walk through the walls. Furthermore, in this topic, students will also add a bomb object – one of the main parts of the Bomberman game – to the project.

Intermediate commit: a23230234d7709456beaf7ec1cc3f1c3150fc108

5.1. Add code for the vertical movement

Analogously, add the code for the up and down directions in the same way (so you will change the value of the local variable **`y`**).

`Commit`: 369070027d0d2eec7e045181e35a3705a9dda367

Local variables will be important as parameters of the **`canEnter()`** method prepared in the following intermediate commit.

Intermediate commit: 3c80533a5a872785cbbd29c3cf1ad4c57f098611

5.2. *Verify movement possibility*

Modify the method that ensures player movement (**`moveUsingArrows`**) to verify the possibility of entering the target cell before changing the player's position.

Commit: 4edb51253991482e1ca431fcb158fefa2eac9de2

In the following intermediate commit, the **`canEnter()`** method is created.

Intermediate commit: 59d511f7170e1c56c97294c65ffbecd32665879d

5.3. *Consider the brick walls*

Modify the **`canEnter()`** method so that the player also reacts to instances of class **`BrickWall`** and cannot pass through them.

Commit: a4757d9ccd7cbc8d5b0241515f268519358e3f61

5.4. *Add a bomb*

Create a new **`Bomb`** class, and design its attributes to represent the force of the explosion. Create a parametric constructor and initialize the attributes of the object.

Commit: 45fe4733e5ffeb58fee0fe2bc9e1d4af8322b0d4

The following intermediate commit adds a bomb placement key to the **`Player`** class. It also adds an attribute for the bomb power of the bomb placement object instance - **`bombPower`**. Thus, a constructor modification is also required.

Intermediate commit: f014c7b3c13416f0f27ae4cbb4c27b039f23f944

5.5. *Check bomb planting possibility*

Add a **`canPlantBomb()`** method to the **`Player`** class with a return value of type **`boolean`**, which returns a flag telling whether it is possible to place a bomb on the cell where the player is currently standing. A bomb can be placed when the appropriate key is pressed and there is no other bomb on the cell.

Commit: ae731b4078f9e4dc19aca31f63badaf068010016

The following intermediate commit adds the **`timer`** and **`owner`** attributes to the **`Bomb`** class. The timer represents the time it takes for the bomb to explode. Therefore, the constructor of the **`Bomb`** class must be modified. The **`act()`** method modifies the behavior of the bomb. When the timer expires, the bomb explodes and disappears from the world. Furthermore, the **`Player`** class is modified. A **`bombCount`** attribute is added to represent the number of bombs that an instance of the **`Player`** class can use. When a bomb is placed, it decreases by 1. When a bomb explodes, it increases by 1. This is handled by the **`bombExploded()`** method.

Intermediate commit: fc8486d412a7c621d2a1796979d110b3b9a7167d

5.6. *Add sound effects*

Extend the game so that the bomb explosion is accompanied by a sound effect. The sound can be recorded or downloaded from the internet. Find the command to play the sound in the Greenfoot class documentation.

Commit: 90da644a58a204eb4de5fb2abab8476c94d73a7c

6. Inheritance and for loop

This topic deals with the introduction of inheritance. The students create a superclass for the classes **`Wall`** and **`BrickWall`**. Then they will create a test arena as a subclass of the **`Arena`** class. Finally, this section focuses on loops with a fixed number of repetitions.

6.1. Add an ancestor class

Create a class **`Obstacle`**. Which class is the superclass of the **`Obstacle`** class? Edit the headers of classes **`BrickWall`** and **`Wall`** so that they are subclasses of the **`Obstacle`** class.

Commit: bb77f8893773fec2853c9518fa1025cf2d6d24e2

6.2. Simplify the occupancy test

After adding the **`Obstacle`** superclass, it is easier to test if the player can enter a given cell. In its **`getObjectsAt()`** method, the world requires as a third parameter a class to look for on a given cell. Since both **`BrickWall`** and **`Wall`** are **`Obstacle`**, it is possible to treat them uniformly. Modify the **`canEnter()`** method in the **`Player`** class to use only a single list of obstacles.

Commit: c3be615a5a3f5f9992f47068d9c5f6d335e52b93

The following intermediate commit shows the creation of the **`TestArena`** class as a subclass of the **`Arena`** class.

Intermediate commit: ba2f48e3a8f38503342d26e9dbae6e3a68f1e2ae

6.3. Modify the arena class

Modify the **`Arena`** class so that its constructor has two parameters representing width and height. Modify the call to the superclass constructor in the **`Arena`** class to take these parameters. Notice, that **`Arena`** cannot be automatically constructed by **`Greenfoot`**, since it needs parameters for constructor Remove from this constructor the code responsible for creating and placing players in the arena, this will be done by subclasses. You can also remove the declaration of attributes of type **`Player`** from the **`Arena`** class.

Commit: 9477fb9718f0269d4025491d9160ad37da8636b7

6.4. Fix the TestArena class

Modify the constructor of the **`TestArena`** class to create an empty arena with a size of 7x7 cells. To verify, create an instance of the **`TestArena`** class – from context menu of class **`TestArena`** select item **`new TestArena()`**.

Commit: 1a27ff1e964c1a2f5820d14b65bc7850562acd01

6.5. Add dimension query

Add a **`showDimensions()`** method to the **`TestArena`** class that prints the arena dimensions to the screen.

Commit: 49cf3931d68f1d3a18df0a7207a2802b90ef54e5

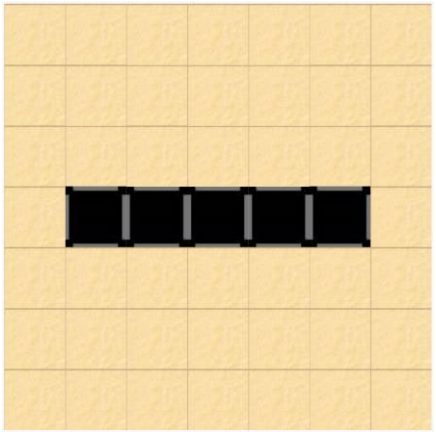
6.6. Add another dimension query

Add a **`showDimensions()`** method to the **`Player`** class that will display the dimensions of the arena if it is in the test arena – class **`TestArena`**. Use the **`showDimensions()`** method of the **`TestArena`** class.

Commit: c5b1e10011483e325271b3a2b101cfbf369f1f44

6.7. Add walls to the test arena

Modify the constructor of the **`TestArena`** class to create a 7x7 cell arena with walls laid out as follows:



Recall that we can use the **`addObject()`** method of the **`World`** class, which has three parameters, to insert an instance of the **`Actor`** class into the world (i.e. into the descendants of the **`World`** class and, in our case, the **`Arena`** class):

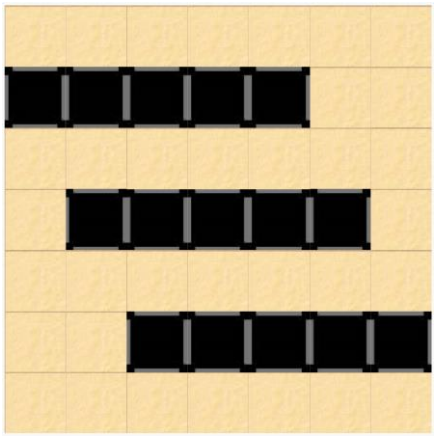
- an actor to be inserted,
- x-coordinate of the cell to be inserted (i.e. column index numbered from 0),
- y-coordinate of the cell to be inserted (i.e. the row index numbered from 0)

The **`[0;0]`** position in the world is at the top left. Since we want to add five instances of the **`Wall`** class at once, we use a **`for`** loop that repeats the statements inside the block for all **`i=1,2,3,4,5`** in our case. Recall that **`super`** is a call to the constructor of the parent class, in our case the **`World`** class. **`Super`** must be first in the constructor

Commit: 2edebe2d623173c4a542a80458e98daf9c043173

6.8. Add even more walls

Modify the constructor of the **`TestArena`** class so that the cells are laid out as shown in the figure.



Commit: a9f8c1af0fb0ce2f15ab4267107bdd87f6a76a73

6.9. Think about how to represent a series of walls

Let's think about how much information we need to be able to create any series of consecutive walls.

6.10. Add a method for the creation of a series of walls

Create a **`createRowOfWalls()`** method in the **`Arena`** superclass, which will have three parameters:

- the row (the top row has index 0) on which to start creating walls,
- the column (the left column has index 0) from which to start creating walls,

- a number expressing how many consecutive walls are to be created.

The method has no return value (so we use the **`void`** keyword).

Commit*:* b155b177619e41170ac7759f8d43ebf748a5da6b

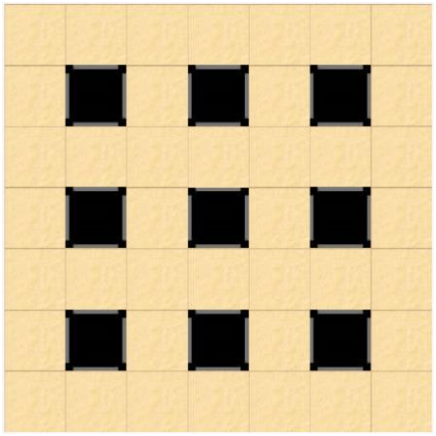
6.11. Use the new method

Modify the code in the constructor of the **`TestArena`** class to use the **`createRowOfWalls()`** method from its superclass – class **`Arena`**.

Commit: 91d8888d11fa710c0c2f7eb8a6e1c9da32139a66

6.12. Change the arrangement of the walls

Modify the constructor of the **`TestArena`** class to create the arena shown in the figure below. To do this, modify the **`createRowOfWalls()`** method to have a fourth parameter defining the spacing between the walls.



Commit: 9f6bd3b3187fb8558b68a58ff7b4a3a56f8b1e82

6.13. Think about how to represent a rectangle of walls

Let's think about how much and what kind of information we need to be able to create walls in a rectangle arrangement whose starting point can be specified and for which the spacing between walls in both rows and columns can be set.

6.14. Add a method creating a rectangle of walls

Declare the **`createRectangleOfWalls()`** method in the **`Arena`** ancestor, which will have the following parameters:

- the row (the top row has index 0) from which to start creating walls,
- the column (the left column has index 0) from which to start creating walls,
- the number of rows to be created,
- the number of consecutive walls to be created in the row,
- the number of empty cells (rows) between the rows,
- the number of empty cells between the walls in the row.

The method has no return value.

Commit: 4abb74dc000ce5f91c05b24ee41bace3b0efc395

6.15. Use the new method

Modify the constructor of the **`TestArena`** class so that it utilizes the method **`createRectangleOfWalls()`** to lay out the arena as shown in Task 6.12.

Commit: 69e31efd4dd9c3a4d44925b0a46eac4f693523dd

6.16. Test your arena

Create several walls in your arena and add two players. Test the functionality of the game, i.e. whether the players will not go into a **`BrickWall`** or a **`Wall`**.

7. List and for each loop

This chapter focuses on lists in more detail using them to track other objects in the arena. It introduces the basic methods for working with a list (creating, adding an element, removing an element, accessing an element) and it also teaches how to use the for each loop to easily access all the elements of a list.

7.1. Check game ending

Create a method **`isGameEnded()`** with no parameters in the **`Arena`** class that detects whether the game has ended (only one or no players are left) and tells whether it has happened in the return value of the **`boolean`** type. For now, let's assume that the end of the game never occurs.

Commit: 4c26e33af1a085e947b1271148b012934520ad9d

7.2. End the game

Add the **`act()`** method to the **`Arena`** class. In the method, check if the game has ended (using the **`isGameEnded()`** method) and if so, stop the game. To stop the Greenfoot environment, use the **`Greenfoot.stop();`** command.

Commit: b58443ab6fe8137801a7b2168cdc9935433e15fe

7.3. Add a list of players

Add the attribute **`listOfPlayers`** of type **`LinkedList<Player>`** to the **`Arena`** class. Don't forget that you have to import the package with the **`LinkedList`** class. Initialize the attribute in the constructor of the **`Arena`** class.

Commit: e48d151c44b93fe44a56ad8af3b41d4c21e10432

7.4. Register the players

Add **`registerPlayer()`** method to the **`Arena`** class that takes a single parameter of type **`Player`** and inserts it at the end of the list of **`listOfPlayers`** using the **`add()`** method. Modify the subclasses of the **`Arena`** class to register the player in the superclass (**`Arena`**) when a player is inserted into the world at the correct location.

Commit: 1166513ff7bfd7d5415715365103821ea0c4691e

7.5. Unregister and remove a player

Add **`unregisterAndRemovePlayer()`** to the **`Arena`** class that takes a single parameter of type **`Player`**. The method removes the player from the player list and then removes the player from the world.

Commit: f7dadce00ade4d12982a0bce2b68e358f448f01b

7.6. Correctly end the game

Implement the body of the **`isGameEnded()`** method so that the method returns true when there is one or no player left in the game. Use appropriate list methods.

Commit: 9ba605f797adec36d7809d6719bc6effa98696c1

7.7. Make bombs dangerous

Modify the code in the **act()** method of the **Bomb** class so that before the bomb is removed from the world, it will kill all players who are at most one power away from it. The **getObjectsInRange()** method returns a list of type **List**. Its first parameter is the range - in this case the force modifier. Its second parameter is identical to the class whose instance it searches within the given range

Commit: c594df2a673abdcadc9ef4161603d54b846e8d62

7.8. Remove affected players

Use the **for** loop to iterate over all the players in the list of players affected by the bomb. Unregister such players in the **Arena** class.

Commit: 2d18cb6f57cf83951cc1a319e5d7f8179508a1a9

The following intermediate commit shows a more efficient way of traversing the list of affected players.

Intermediate commit: 850919027622ff95e0985c6eae9409b96337d152

7.9. Edit the player class

Create a **hit()** method in the **Player** class that will be invoked by the player's bomb after the bomb hits him. Unregister the player from the world in this method. Edit the code in the method **act()** of the **Bomb** class to reflect the new functionality.

Commit: f803a0f8bfddc3ab6f7eef45817d482e11554c63

7.10. Remove the owner

Create a **removeOwner()** method in the **Bomb** class that sets its owner attribute to **null**. An object instance can be thought of as a pointer - an "arrow" to the object. By setting it to **null** the pointer points to no object.

Commit: 3a20a51c04cb0ea2b47cf789c2b6503e3ab86eab

At the moment we have set the **owner** attribute to **null**. So the attribute does not need to point to an instance of the object. Therefore, before calling the **bombExploded()** method, we need to check if its owner exists. The situation is resolved in the following intermediate commit.

Intermediate commit: a7e97fa06d4c2ab7a39967a80a0410c30bd28707

7.11. Add a list of bombs

Create an attribute **listOfActiveBombs** of type **LinkedList<Bomb>** in the class **Player**. Initialize it in the correct constructor. Modify the method bodies according to the following rules:

- in the method **act()** register a newly created bomb to the **listOfActiveBombs**;
- in the **bombExploded()** method, remove the bomb that came as a parameter (the one that exploded) from the **listOfActiveBombs**;
- in the **hit()** method, use the for each loop to remove the owner from all the bombs in the **listOfActiveBombs**.

Commit: 85e552712133e34b0422ebfd4a510147d6d3e027

8. Private methods and while loop

This topic introduces the **while** loop – a loop that repeats while some condition defined at the beginning of the loop evaluates to **true**. Furthermore, it also teaches the students how to create

private methods – methods that can only be invoked by an instance of the class in which the method is defined.

8.1. Create the fire class

Create the **`Fire`** class. Choose an appropriate graphical representation. The constructor of this class has one parameter that determines how long the fire burns in place. Ensure that the fire disappears from the world after a given amount of time.

Commit: a68bf80f0af90da351b090217414c63e7fe9492b

8.2. Start the fire

Modify the existing bomb explosion code to leave an instance of the **`Fire`** class at the bomb site. Test your solution.

Commit: ad9cbedf491e77a9d2ea8900bcb3a894b681997c

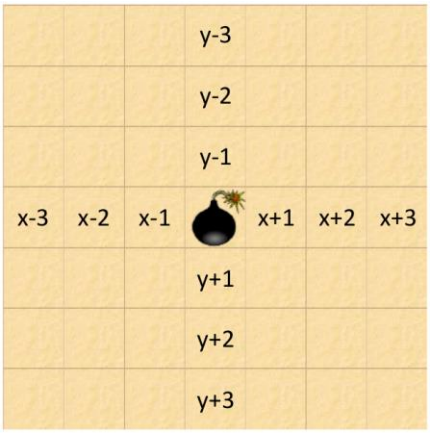
8.3. Spread the fire

Use the **`for`** loop to extend the bomb explosion (create an instance of the **`Fire`** class) in the right direction from the bomb. Extend the blast to as many cells as indicated by the bomb's strength attribute.

Commit: acf9745908e32a21f43265a4a17662aa06516778

8.4. Spread the fire in all directions

Adjust the bomb blast so that it generates fires in all directions. Help yourself by changing the coordinates as shown in the image below.



Commit: 7f1b51fc85583b89e208051958b07d22753977db

8.5. Rewrite the loops

Rewrite all **`for`** loops for fire propagation using the **`while`** loop. Omit the second part of the condition (it is possible to put fire in the next cell) for now.

Commit: 3e34c8950afe001ceb37e768230a7b91d7ee6cb9

In the following intermediate commit, the **`spreadFire()`** method is created with two parameters. Walk through the method and describe where it will set the fire depending on it's input parameters. The method is **`private`**. This means that it can only be called from the **`Bomb`** class.

Intermediate commit: b11aa272c157dbf3c16f9dd6e8e5fa9ad6542ea1

8.6. Use the private method

Use the private method **`spreadFire()`** to spread the fire after a bomb explosion.

Commit: 9f81a2971a3312ec0883fb2f795d8301b47bb18c

8.7. Add another private method

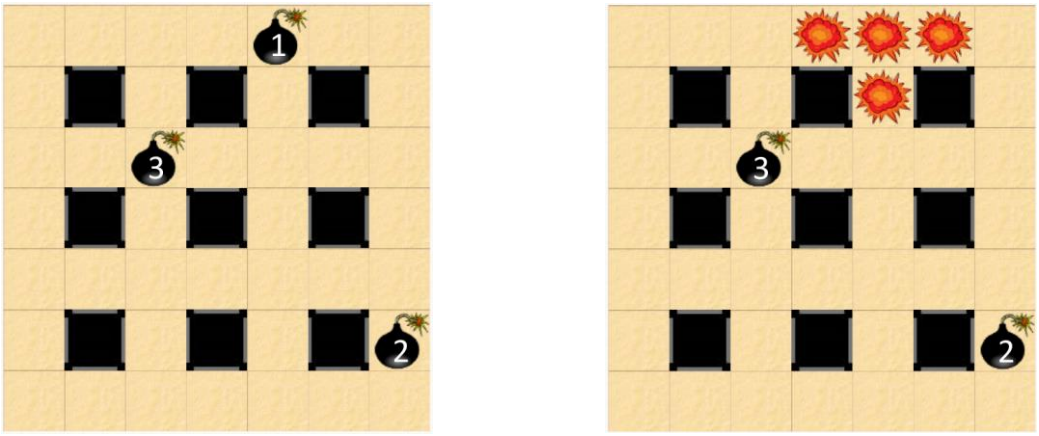
Create a private method **`canCellExplode()`** in the **`Bomb`** class that takes row and column coordinates as the parameters and returns **`true`** if an explosion can occur on that cell and **`false`** otherwise. The fire cannot continue if:

- it reached the edge of the world,
- if the cell contains a wall.

Commit: 53af76996cf6c8e4456627e0cc7bcadd79ab80fa

8.8. Use the private methods

Using the method **`canCellExplode()`** you can now modify the condition in the **`while`** loop in the **`spreadFire()`** method in the **`Bomb`** class. Modify the condition in the loop to respect the result of the check from the **`canBombExplode()`** method. Test the functionality of the solution with bombs of different strengths between the walls. Tests of different explosions are shown in the following figures:



a *b*

c *d*

In part (a) we can see the original distribution, in part (b) a bomb with force 1 exploded, in part (c) a bomb with force 2 exploded and in part (d) a bomb with force 3 exploded.

Commit: bb7fd73866546b94537195b58b119ab62b31dc9e

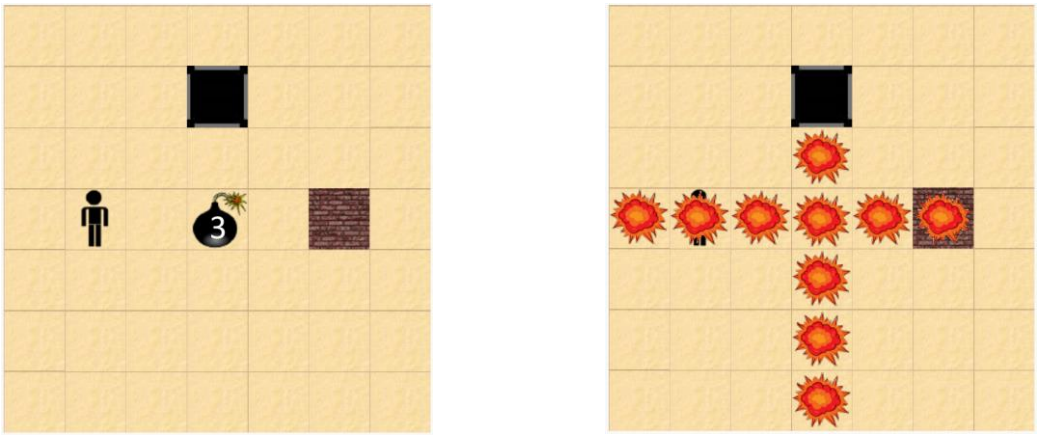
8.9. Check explosion obstacle

Add **`canExplosionContinue()`** private method to the **`Bomb`** class which takes row and column coordinates in parameters and returns **`true`** if the cell did not stop the explosion at the given coordinates. If the cell has stopped the explosion, the method returns **`false`**. The explosion cannot continue if it has hit a wall.

Commit: f8d3838b42f0b9ac2984f4275159d5d10a059ebb

8.10. Limit the explosion

Using the **`canExplosionContinue()`** method, modify the **`spreadFire()`** method in the **`Bomb`** class. If the explosion can continue from the given cell, increase the value of the variable **`i`** by 1 and recalculate the coordinates of the new row and column of the explosion. Otherwise, artificially increase the value of variable **`i`** to a value greater than the bomb's force, which stops the loop. Test your solution on the situation from the following figure:



A *b*

Commit: eeb5e1cb2a887badad6aa8742a0f1fac901ff24a

As you can see in the picture, the brick wall explodes, but stops the explosion. So it should disappear after the explosion. This is solved by the following intermediate commit.

Intermediate commit: 91f2be5f9b170744027ce763793d527c4ad9c2c4

8.11. Modify the fire presence check

Modify the behavior of the **`Bomb`** class instance so that it does not call the player's **`hit()`** method in its reach. Instead, the player itself will check for overlapping fire. Modify the behavior of the player's **`act()`** method so that it first checks to see if it overlaps with an instance of the **`Fire`** class. If it does, it will call its **`hit()`** method itself. This will ensure that the player is also hit by the fire that burns after the bomb explodes.

Commit: 13acf174668443d98eec3de6e68615ee90b4fd9d

8.12. Add a chain of explosions

Modify the **`act()`** method of the **`Bomb`** class so that the bomb explodes even if it is in the same cell as the fire. Verify the solution by doing a chain reaction of several bombs.

Commit: 137f5de9a734561a11ea0afc6cfd5dde5e1ee1bd

9. Polymorphism

The goal of this topic is to teach students how to create virtual methods and overlay them as needed in subclasses. Also, the students will be introduced to a new visibility for attributes and methods – the **`protected`** access modifier. The students will use polymorphism to simplify existing code.

9.1. Add mines

Let's start by adding a mine. The mine explodes just as the player steps on it. A mine will also always explode when it is hit by fire (e.g. from a bomb that exploded nearby). It leaves fire in its place (and only there). Add the ability for the player to lay mines (like bombs) when a key is pressed (e.g. **`control`** or **`shift`**). Similar to bombs, the player also has a limited number of mines (i.e. if he lays all mines, he can only lay another mine if one of the previously laid mines explodes). The initial number of mines is set by a parameter of the **`Player`**'s constructor. To register mines and react to their explosion in the **`Player`** class, follow the same procedure as for bombs (create a list of mines, add methods **`mineExploded()`**, **`canPlantMine()`**, etc.).

Commit: 5627d97dff9b5d5c1533d9cfb1a3f4a7c8bf631d

9.2. Add superclass

Create a common superclass for the **`Bomb`** and **`Mine`** classes – the **`Explosive`** class. Which attributes and methods should be moved to the superclass, and which should remain in the subclasses? Modify existing classes according to your design.

Commit: 48eb9fb40e332e027c2e2b168a1d6d3149d7fd3d

9.3. Adjust attribute visibility

Modify the visibility of the **`owner`** attribute in the **`Explosive`** class to **`protected`**. The visibility of the **`private`** attribute would allow it to be used only inside the **`Explosive`** class and not in its descendants. The **`protected`** means visibility within the package, which in our case is within the Bomberman project.

Commit: 5a0b733d1df877777bad7688631f1500cd8c7041

9.4. Add text output

Create a **`printWhoYouAre()`** method with no parameters in the **`Explosive`** class that does not have a return value. The method prints the text “EXPLOSIVE” to the screen where the explosive is currently located. Create an instance of the **`Mine`** class and call the **`printWhoYouAre()`** method. What happens? Create an instance of the **`Bomb`** class and call the **`printWhoYouAre()`** method. What happens in this case?

Commit: 0528c0a1ec75f08b2f2d088314432c5169448169

9.5. Improve the text output

Create a **`printWhoYouAre()`** method in the **`Mine`** with the same header as in the class **`Explosive`** (i.e. the method will have the same name, the same parameters, and the same return value type). The method will print the text “MINE” to the screen. Again, create an instance of the **`Mine`** class and the **`Bomb`** class. Try to guess what happens when you call the test method in an instance of class **`Mine`** and an instance of class **`Bomb`**. After that, call the methods. Does your guess match the result?

Commit: 2d3e1fe13a8d627730bf01a729ae7c1619eaf64b

9.6. Finalize the text output

Override the **`printWhoYouAre()`** method in the **`Bomb`** class so that it writes the text “BOMB” to the screen. Verify the correctness of your solution.

Commit: 09e622e84efe580fed9c0b67f2bdd2a48cd06b1f

9.7. Add explosion handling

Create **`shouldExplode()`** and **`explosion()`** methods in the **`Explosive`** class and **`explosiveExploded()`** method in the **`Player`** class as described above. Do not implement the method bodies yet. If a return value is needed, return **`false`**.

Commit: 749e2e91092f8929877af7d8aeb068f010b110ed

9.8. Make the explosive explode

Write the body of the **`act()`** method in the **`Explosive`** class.

Commit: 996cfa17d46269663becddd1e1fbe2fb82280370

9.9. Make the bomb explode

Override the **`shouldExplode()`** and **`explosion()`** methods in the **`Bomb`** class. Use the appropriate code from its **`act()`** method. Notice that it is possible to simply write the bodies of the methods since there is no need to reason over the conditions (the superclass did that).

Commit: c1eed984efd956ff4ddd914288201773afd92d44

Just as **`super`** was used for constructor to call the parent constructor, this is possible for other methods as well. The following intermediate commit shows a call to the **`act()`** method in the **`Bomb`** class from the **`Explosive`** class using **`super`**.

Intermediate commit: 5a6d0ceaa51024854ac579f1c3a888309914e22a

9.10. Make the mine explode

Override the **`shouldExplode()`** and **`explosion()`** methods in the **`Mine`** class. Use the appropriate code from its **`act()`** method. Why is it necessary to remove the **`act()`** method in the end?

Commit: f3d63427a9a975f5205e483ea535c4440582de28

9.11. Add interaction with fire

Modify the body of the **`shouldExplode()`** method in the **`Explosive`** class so that the method returns **`true`** if the instance touches an instance of the **`Fire`** class. Modify the overridden methods **`shouldExplode()`** in the **`Bomb`** class and the **`Mine`** class to use the functionality of the ancestor method.

Commit: 82767b061d4e2e48810f57133a090c2b8d98017d

9.12. Simplify player attributes

Remove the attributes **`listOfActiveBombs`** and **`listOfActiveMines`** from the **`Player`** class. Add a single attribute of type **`listOfActiveExplosives`** of type **`LinkedList<Explosive>`** to the **`Player`** class. Initialize it in the constructor and remove the initialization of the original attributes from the constructor.

Commit: c174bfb8c5512cb2e9e1fb45ecf36c83d84b5af7

In connection with the previous change, the **`Player`** class needs to be modified wherever the lists **`listOfActiveBombs`** and **`listOfActiveMines`** were used. They need to be replaced by the **`listOfActiveExplosives`** lists. Everything is resolved in the following intermediate commit.

Intermediate commit: ce7945c389bfd55b9eabf76c4599bc83f273e07e

9.13. Simplify explosion handling

Implement the body of the **`explosiveExploded()`** method. Using the **`instanceof`** operator to determine if the explosive is a **`Bomb`** or if the explosive is a **`Mine`**. Based on its actual type, increment the counter of available bombs or the counter of available mines. Be sure to remove the explosive from the list of active explosives. Finally, remove the unnecessary **`bombExploded()`** and **`mineExploded()`** methods.

Commit: 7a50f76f5de55cad056d372563214f0cbf581c97

10. Random numbers

This topic is devoted to randomness. The students will learn about the **`Random`** class. With the help of its instances, they will generate random numbers. The topic also shows a way of generating random numbers without using the **`Random`** class by directly using the **`Greenfoot`** environment. The students will use random numbers to randomize the layout of the arena, and to add bonuses to the world – special elements that are created after the brick wall explodes, and that improve selected player properties.

10.1. Think about randomness

Think about what randomness is, how we can get some random result from an experiment, and what random phenomena we observe in the world around us.

10.2. Think about generating random values

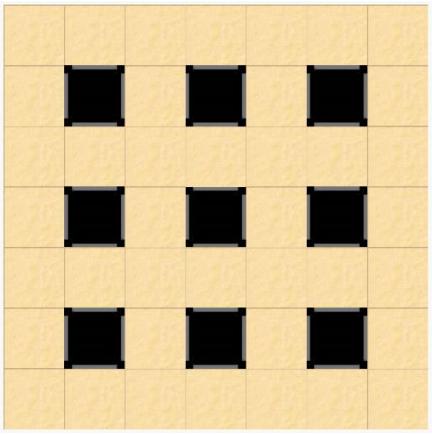
Let us consider a classic dice with six sides. Could we use it to generate a random position on a chessboard with 6x6 squares? What about a chessboard with 3x3 squares? How would the method of generation change if we used a coin? Suggest such position generation algorithms.

10.3. Observe randomness

Using the algorithm from the previous task and with the help of a die, generate random positions on the chessboard. Record your results on the chessboard. Can any regularity in the results be observed?

10.4. Prepare a random arena

Prepare an arena. Create a subclass of the **`Arena`** class, which you name e.g. **`RandomArena`**. Set the appropriate world size in the constructor. We recommend a regular wall layout with one empty field between the walls as shown in the following figure:



Commit: f937ffbd36f64b0df7c9d904e974a2437d9c8959

In the intermediate commit there is empty method **`createRandomWall()`** prepared.

Intermediate commit: f692394053ce0a3f76c6ec294c977ef3d7dba91c

10.5. Add a random number generator

Add a reference attribute of type **`Random`** to the **`RandomArena`** class. Remember that the class **`Random`** is defined in the package **`java.util.Random`**. Initialize the attribute in the constructor.

Commit: 69655b53dac3fe0565d88e124cab63a03ee5b533

In the following intermediate commit, random numbers are generated and stored in the **`randomColumn`** and **`randomRow`** variables.

Intermediate commit: 1ba83506b6b2ea5b81ba79c67c506a81d11e8dc5

10.6. Check cell occupancy

Add an **`isCellFree()`** private method to the **`RandomArena`** class that takes two parameters – column and row. The method will return **`true`** if the cell is free in the world (contains no instance of class **`Actor`**), otherwise, it will return **`false`**.

Commit: 05dc486ffc38a6dd69e510072ce15d0e9ad4698a

Now it is possible to generate the coordinates of a cell where there is no object of the **`Actor`** class and insert an instance of the **`BrickWall`** class into it. Everything is solved by the while loop in the following intermediate commit.

Intermediate commit: 6507a2386b739c1894f96cba24d49661d0195571

10.7. Generate random walls

Modify the constructor of the **`RandomArena`** class to randomly generate walls into a third of all cells in the arena.

Commit: e92647d730fb61f573a144c06c690d9366889869

10.8. Move randomness to the ancestor

Move the **`createRandomWall()`**, **`isCellFree()`** methods, and the generator attribute (including its initialization in the constructor) to the **`Arena`** superclass. Don't forget to move the import lines as well.

Commit: bd5f0d27c3437995362161e948dbf2af381e8715

10.9. Generalize random generation

Add a parameter of type **`Actor`** to the **`createRandomWall()`** method – this will be the actor, which we will insert at random coordinates. Change the method name (e.g. **`insertActorRandomly()`**) and update the method call from the **`RandomArena`** class.

Commit: 55cf63e631abcc7924e83fa38d558e3bd8e1f8aa

10.10. Add bonuses

Create a **`Bonus`** class as a subclass of the **`Actor`** class. Create two subclasses of the **`Bonus`** class – class **`BonusFire`** and class **`BonusBomb`**. Set appropriate images for the classes.

Commit: df2b5f48f12f9cdc1de0528422dbda07e5051d59

10.11. Create bonuses randomly

Edit the code in the **`act()`** method of the **`BrickWall`** class. After it is destroyed, generate in its place with a probability of 10 % a bonus fire, and with the probability of 10%, generate a bonus bomb. In 80% of cases, nothing is generated after destruction.

Commit: 7ec336a713814cdae7fe1be841c5e4feacd75e74

10.12. Apply the bonus

Prepare the **`Bonus`** class. Create a **`protected applyYourself()`** method with no return value, which takes a single parameter of type **`Player`**. Leave this method empty in the **`Bonus`** class. Then define an action in the **`act()`** method to first detect if a player has stepped on the bonus (method **`(Player)this.getOneIntersectingObject(Player.class)`**) and if so, apply it (by calling the virtual method) and finally remove the bonus from the world.

Commit: 684b7d72b6d5c22a355ae7ca76e53f8cc540908d

10.13. Increase bomb count

Add a **`public increaseBombCount()`** method with no parameter to the **`Player`** class that does not have a return value, and which will increase the number of bombs the player can plant by one. Then override the **`applyYourself()`** method in the **`BonusBomb`** class. Applying the bonus means increasing the number of bombs the player has by one (calling the **`increaseBombCount()`** method).

Commit: cb43a15261ded0f12df3b92cfe5d50efe24e9e95

10.14. Increase bomb power

Add a **`public`**, **`increaseBombPower()`** method with no parameter to the **`Player`** class that does not have a return value, and which increments the value of the **`bombPower`** attribute by one. Then override the the **`applyYourself()`** method in the **`BonusFire`** class and increase the player's bomb strength by one.

Commit: 4abf63b01f1eee1c870821a90cb0e920594443af

3.2. Tower defense

In tower defense like games, player uses towers shooting some kind of bullets to stop enemies from reaching and destroying an orb. The enemies always follow the same path, however as the game continues, the enemies become stronger and spawn in larger groups. The player must place the towers in strategic places to stop them in upcoming waves. There exist many versions of the game that take place in different worlds using different entities (from balloons up to orcs, defending using animal like towers up to magical pools).

This project will introduce one type of tower that can or cannot be manually controlled by the player. Enemies will be of different types with differences in their HP and speed. The introduced game design is easy to expand, what leaves enough space for students' creativity as well as for teacher's assignments. For this reason, we tried to minimize assessment type activities in the first chapters. Moreover, the design leaves enough space to naturally introduce topics out of scope of light OOP (such as polymorphism) with ease. The project in its final state during gameplay is shown in Figure 3.

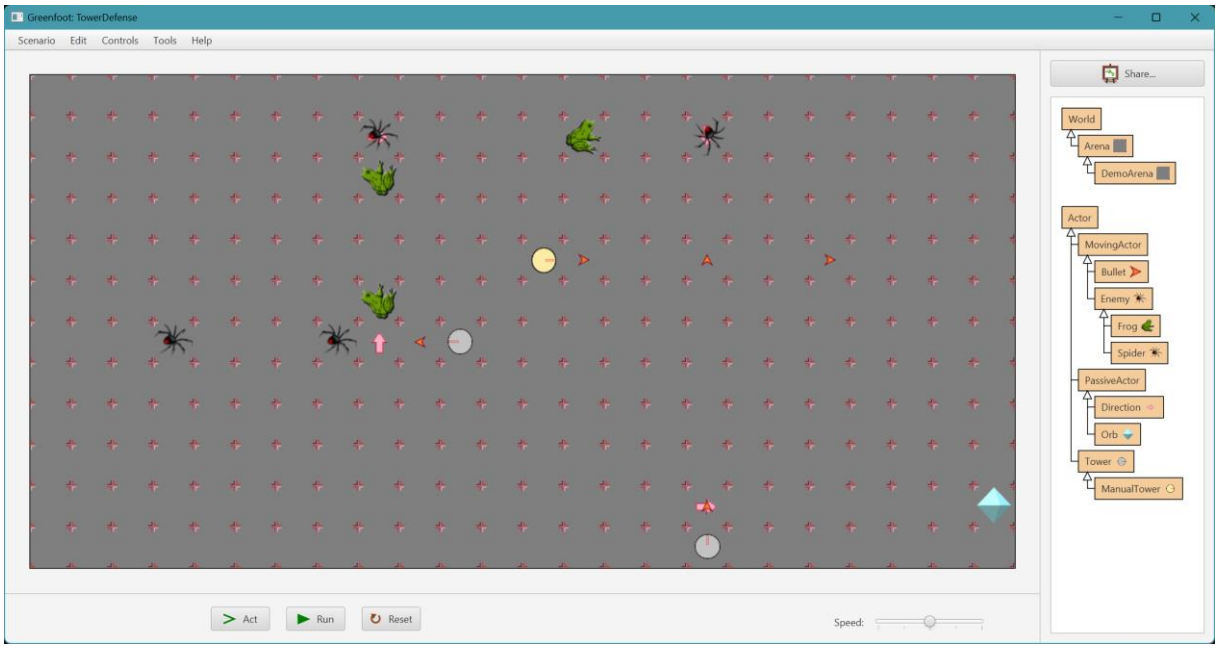


Figure 3: Greenfoot environment with final state of project Tower defense

Source codes are available at:
https://gitlab.kicon.fri.uniza.sk/oop4fun/project-tower-defense

Learning design is available at:
http://learning-design.eu/en/preview/452257b563cbf14b6f06acfd/details

3.2.1. Content and scope of the educational program

Overall learner workload is 33h 5min and is distributed as follows:

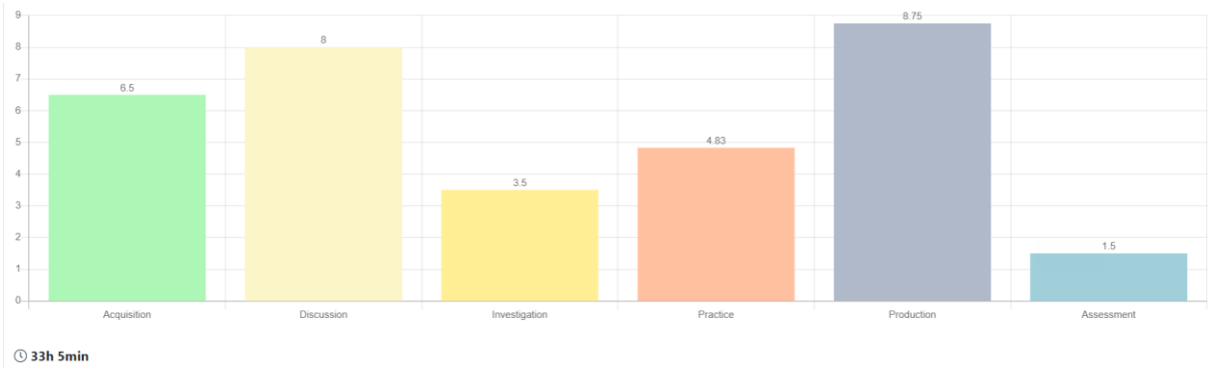


Figure 4: Learner workload when using project Tower defense

Constructive alignment is summarized in table below:

Table 2: Constructive alignment of project Tower defense

Topic	Assessment		Understanding the basic principles of object-orien... (25)	Understanding the basics of algorithmisation (25)	Understanding the syntax of the Java programming l... (10)	Analysing program execution based on the source co... (20)	The ability of creating own programs with the use ... (20)
	Formative	Summative					
Greenfoot environment	0	0					100%
Class definition	0	0	60%		20%		20%
Algorithm	0	0		60%	10%	20%	10%
Branching	0	0	10%	60%	10%	10%	10%
Variables and expressions	0	0	40%	30%	20%		10%
Association	0	60	30%	30%	10%		30%
Inheritance	0	30	40%	20%	10%		30%
Encapsulation	0	0	50%	10%	20%		20%
Total	**0**	**90**	**230%**	**210%**	**100%**	**30%**	**230%**
	90						

For a detailed plan refer to attachment 5.2.

3.2.2. Topics

Project tower defense is divided into seven topics:

Covered topics of light OOP are:

- classes, objects, instance
- methods, passing methods arguments
- constructors
- attributes
- static variables and methods
- encapsulation

- inheritance
- abstract classes
- object live cycle

1. Introduction to the Greenfoot environment

The topic is devoted to project creation. Students will be capable of creating a new project in Greenfoot environment, create class (as subclass of **`Actor`**), select image for newly created class, create its instance and send a message to it.

Create a new project. Give it a proper name (e.g. **`TowerDefense`**) and save it to a proper location.

Commit: 9046f5353d857dcc112abd92d7b7170abcc64a80

Table 3 summarizes comparison of workloads of topic Greenfoot environment between projects Bomberman and Tower defense. There is no difference in the design of the topics.

Table 3: Comparision of workloads of topic Greenfoot environment between projects Bomberman and Tower defense

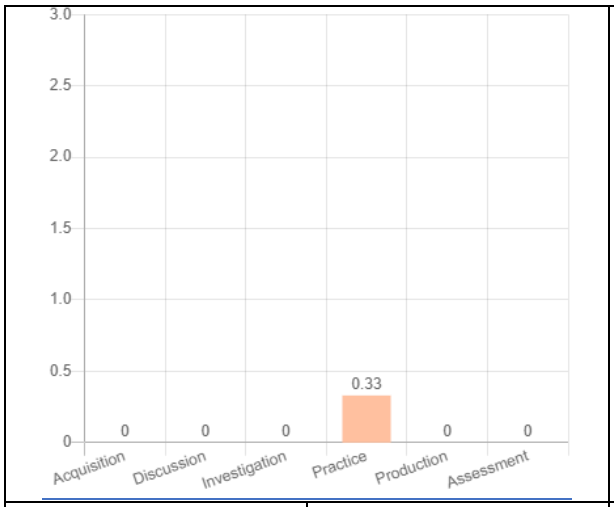
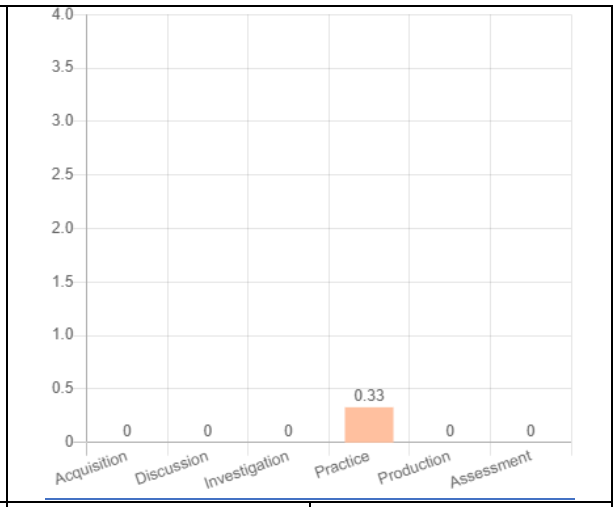
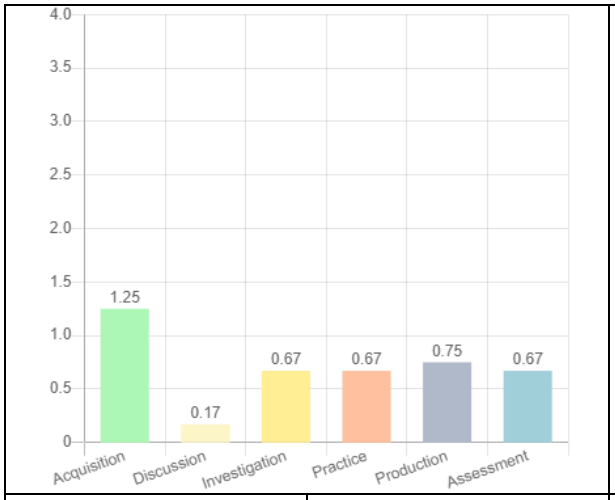
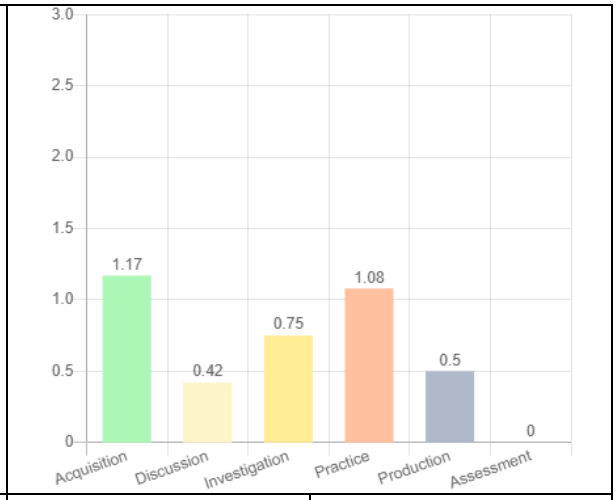
			
Bomberman	0h 20min	Tower defense	0h 20min

Table 4 summarizes comparison of workloads of topic Class definition between projects Bomberman and Tower defense. The workload of Tower defense project is lower, more practically oriented with emphasis on investigation and practice.

Table 4: Comparision of workloads of topic Class definition between projects Bomberman and Tower defense

	Bomberman	4h 10min		Tower defense	3h 55min

1.1. Task 1.1 Identify objects from project Bomberman.

1.2. Prepare the world

Edit source code of class `MyWorld` (double-click on it) to create a world of size 24x12 cells. Each cell should be 50 pixels in size.

Commit: a593cd4a92d0fa0db78275614c3e41a2e96b4e57

1.3. Prepare world graphics

Find or create a proper image for the world background. You may either use prepared images (select item `Set image...` from the context menu of class `MyWorld`) or custom image (copy image into subfolder `images` of your project folder and select it using the same way as described before).

As a background you may use sole image that will cover whole world's area (compute needed size of the image with regards to the world's size) or smaller one, that will be repeatedly copied (use square image with the size of the cell).

Commit: 1184980643db082cfdd6bde9984bceaddf010d49

1.4. Create class Enemy

Create an enemy. Enemy will march towards player's orb to damage and eventually destroy it. Create a new subclass of class Actor (select item `New subclass...` from the context menu of class `Actor`). Give it proper name (`Enemy`) and image.

Commit: 4981400623729c3d112b54454b6e6151e18426bf

1.5. Create instance of class Enemy

Create an instance of class Enemy (select item `new Enemy()` from the context menu of class `Enemy`, put instance into world by left mouse click on desired position). Investigate its internal state (select item `Inspect` from the context menu of the created instance).

Create another instance of **`class Enemy`** and put it in another position. Compare internal states of two created instances.

1.6. Send messages to instance

Send a message to instance of class `Enemy` (select item **`inherited from Actor`** from the context menu of chosen instance and then select desired item). What happened? How was the internal state of respective instance affected?

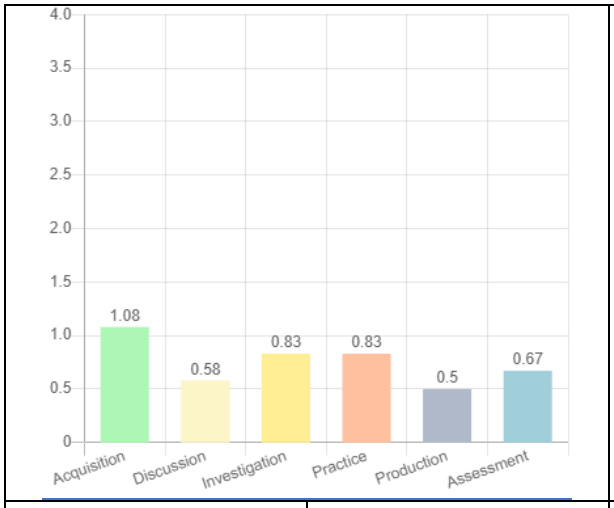
Send messages to the instance of class `Enemy` so it will move into position [12, 6] and it will be facing down. Write the sequence of sent messages onto paper.

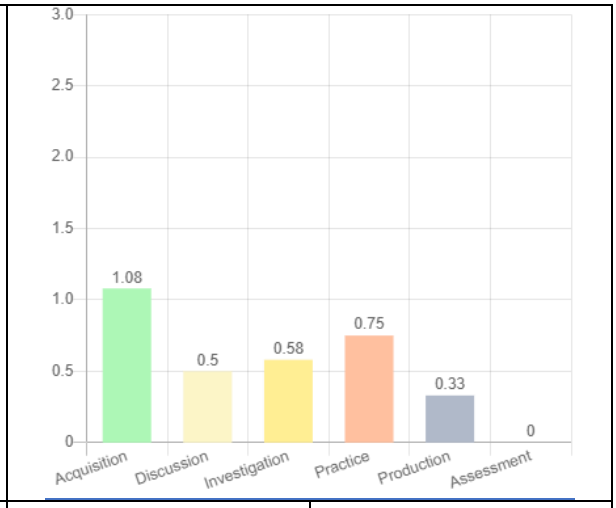
2. Algorithm, application controls, method creation

Topic deals with basics of algorithmization and introduces the work with documentation from very beginning. Students will be capable of calling a method in source code, write and invoke documentation.

Table 5 summarizes comparison of workloads of topic Algorithm between projects Bomberman and Tower defense. The design of Tower defense is similar to Bomberman, however with significantly lower number of TLAs of investigation type. See, that many TLAs are the same as in Bomberman project. This is because in this state of the projects, there is a large overlap of what can be done with it. It is possible to find inspiration in tasks in Bomberman project, if there will be any need to strengthen the investigation part of the syllabus. However, for the sake of teaching light OOP using the Tower defense project, we find the proposed amount of investigation type TLAs sufficient.

Table 5: Comparision of workloads of topic Algorithm between projects Bomberman and Tower defense





Bomberman	4h 30min	Tower defense	3h 15min

2.1. Task 2.1 Write a simple algorithm from project Bomberman.

2.2. Task 2.2 Write a more general algorithm from project Bomberman.

2.3. Call a method

Add a statement to the body of the **`act()`** method such that the instance of the **`Enemy`** class moves two cells in the current direction. Then create more instances of the **`Enemy`** class and call the method on each instance. Is the behavior expected?

Commit*:* 7ba327ebeba6a13be68d9d21cc7e74b0da376132

2.4. Add documentation

Add a documentation comment for the **`act()`** method.

Commit*:* 68b1c82c7df2c7826f2d3f78373498569adab7e9

2.5. Add more documentation

Edit the documentation comment of the **Enemy** class. Add the version of the class and its author.

Commit: 1a7a9f83c5271a7c0dfa46ce3b1ee65682b0c5e5

2.6. Task 2.7 Read the documentation from project Bomberman

2.7. Task 2.9 Explore application controls from project Bomberman

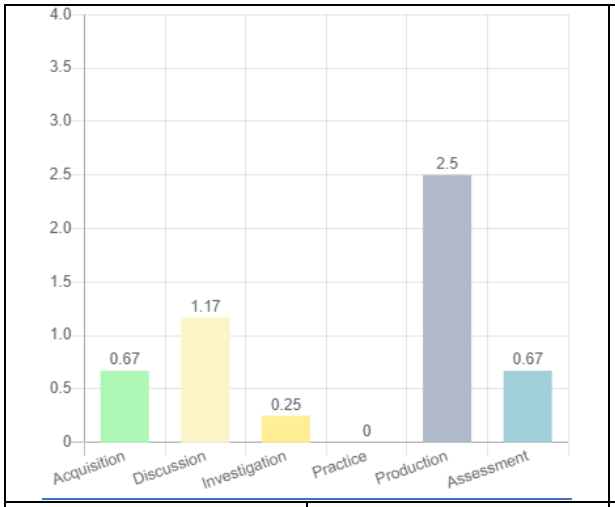
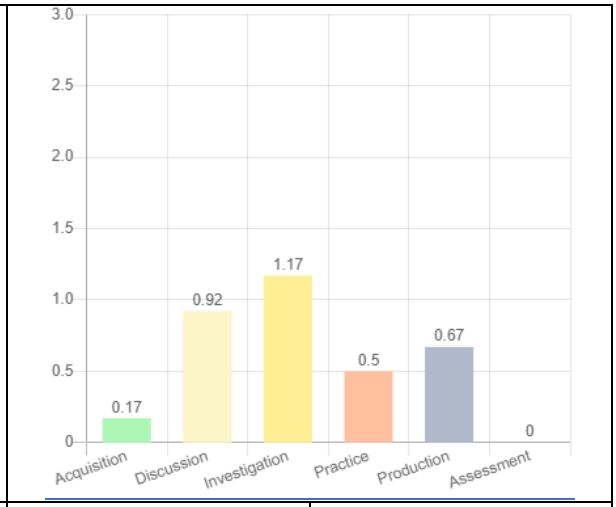
3. Branching and enemy control

Topic covers incomplete and complete branching. Basics of **Actor**'s **World** perception is introduced. Students will be capable of writing code using conditions.

The state of the project in this chapter opens possibilities for teacher to assign tasks like "use instances class of **Direction** and **Orb** to navigate **Enemy** so that it will move in desired pattern", with limit conditions like using maximum number of instances of particular class or tasks like "predict movement in desired setup of world".

Table 6 summarizes comparison of workloads of topic Branching between projects Bomberman and Tower defense. There is a clear difference between the designs. Tower defense splits the production type TLAs to strengthen investigation and practice. This allows to experiment more and leaves the door open for students creativity. Note the low number of acquisition TLAs. This is because a) there is no multiple branching introduced and b) investigation TLAs are emphasized.

Table 6: Comparision of workloads of topic Algorithm between projects Bomberman and Tower defense

	Bomberman	5h 15min		Tower defense	3h 25min

3.1. Observe the enemy state

Create an instance of the **Enemy** class and place it in the center of the board. Open a window with the internal state of the instance and position it so that it is visible while the application is running. Then run the application and observe how the values of the **x**, **y** and **rotation** attributes in the **Enemy** class change. How do these values change as you move (up, down, left, and right) and turn?

3.2. Add world edge detection

Add code to the body of the **act()** method to rotate the enemy 180° after reaching the edge of the world.

Commit: 4927c3ff7eb39b51ba2738f2ab500fd6c32e3bb4

3.3. Add classes Direction and Orb

Create two new classes, descendants of the **`Actor`** class. The first class will be **`Direction`** class and the second class will be **`Orb`**. Prepare suitable (max. 50x50 pixel) images in a graphical editor. Then assign these images to the newly created classes.

Commit*:* 4ed6b37e6d481181d8b340639aa03391406b6c2e

3.4. Add collision detection

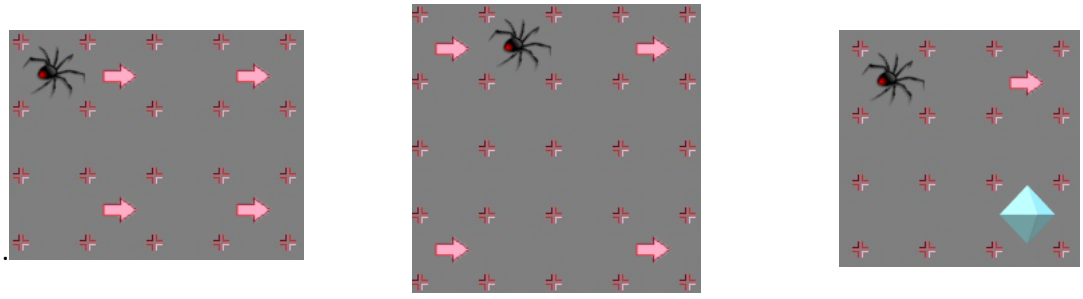
Add code to the **`act()`** method of the **`Enemy`** class to ensure that:

- the enemy turns 90° clockwise when he enters a cell the contains an instance of the **`Direction`** class,
- the enemy turns 90° counterclockwise when he enters a cell the contains an instance of the **`Orb`** class.

Commit*:* 968e6f195e3def25e11bc41b664ba1715f7da11d

3.5. Predict enemy movement on custom setup

Prepare different configurations, inspiration can be found in the figures below. Guess how the enemy will move? Run the application. Does your prediction match what you observe? What caused differences in prediction and reality?



a) b) c)

Figure 5: Configurations of custom setups of instances to predict movement of instance of class Enemy

3.6. Predict enemy movement on specific setup

Prepare the situation as depicted in the figure below. Guess how the enemy will move? Run the application. Does your prediction match what you observe? What caused differences in prediction and reality?

a) b) c)

Figure 6: Configurations of tricky setups of instances to predict movement of instance of class Enemy

3.7. Use full branching with collision detection

From the last chapter, you can see that the problem occurs when an instance of the **`Orb`** or **`Direction`** class is on the edge of the world, or both instances are in the same cell. When branching is incomplete, collisions or repeated rotations occur. Basically, multiple conditions are satisfied at the same time. Therefore, you need to change the code of the act() method of the **`Enemy`** class so that only one rotation

occurs. It is necessary to use full branching. Create a cascade of conditions. The most important check (that is, the first one) is edge detection. The second most important check is the touch check for an instance of the **`Direction`** class. The last is the check for touching an instance of the **`Orb`** class. Modify the **`act()`** method according to these rules.

Commit*:* f017de8b49d4fc77f62afac4d842429560bcfb8b

3.8. Predict enemy movement on previous setups

Run through tasks 3.5 and 3.6 again. What changed?

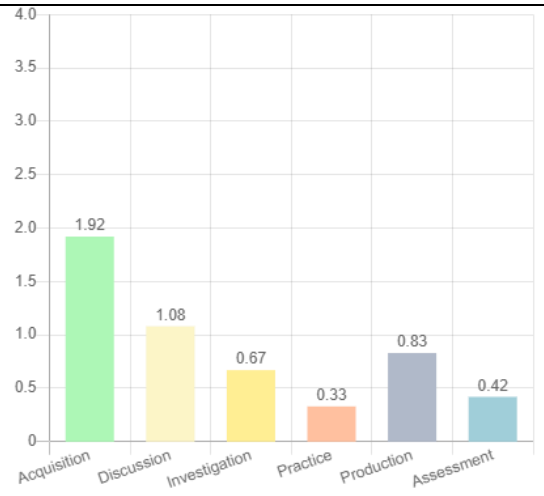
4. Variables and expressions

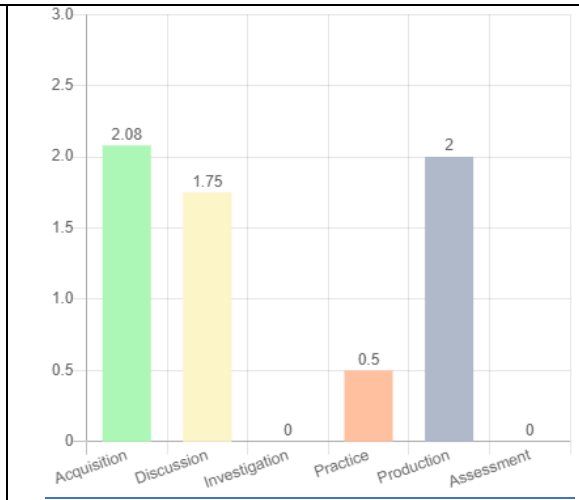
This chapter introduces variables and expressions.

The state of the project in this chapter opens similar possibilities as in the last chapter. With a little creativity there may be added more classes like the **`Direction`** class, that will cause different behavior when step onto (e.g. teleports, tunnels, etc.) by instance of class **`Enemy`**. These classes can be discussed with students and the respective implementation may be assigned as home assignments.

Table 7 summarizes comparison of workloads of topic Variables and expressions between projects Bomberman and Tower defense. See the similarity between designs of previous and current topics between the two projects. This topic is more production oriented (similarly to previous topic in Bomberman) and vice versa (note that Bomberman is different type of project, where student's creativity was with benefits used in this topic).

Table 7: Comparision of workloads of topic Variables and expressions between projects Bomberman and Tower defense





Bomberman	5h 15min	Tower defense	6h 20min

4.1. Turn in direction

Change code in **`Enemy`**'s **`act()`** method so it will turn into the same direction as **`Direction`** class instance (they will have the same rotation). Use **`getOneIntersectingObject(_cls_)`** method to store instance in proper local variable (**`Direction direction`** - you will need to use typecast, since return value is of type **`Actor`**, write **`(Direction)`** in front of **`getOneIntersectingObject`** method call, we will get to typecast later). If any instance of respective class was obtained, extract the **`direction`**'s rotation using **`getDirection()`** method (store it if you want) and then set it into enemy (**`this`**) using method **`setDirection(int)`**. Test your solution.

Commit*:* 97dddc4beba40ac785c7413bb245ba849cd956d2

4.2. Rename class MyWorld to Arena

Give class **MyWorld** better name. Rename it to **Arena**. Do not forget to rename the constructor accordingly.

Commit: aaf73c9bfd9f76a2a1e504f5e78d2976f1cada12

4.3. Create layout of Arena

Create custom layout of **Arena**. Fill the constructor of the class. Add one instance of **Enemy**, one instance of **Orb** and at least one instance of **Direction**. To add (subclass of) **Actor**, you can use following template:

1. Declare and initialize variable of required type (subclass of Actor)
   `Enemy e = new Enemy();`
2. Assign properties using proper methods
   `e.setRotation(90);`
3. Put it into world's (**Arena**) using method **addObject(Actor)**.
   `this.addObject(e, 6, 0);`

Test your solution.

Commit: 8b105ea2eaf697f08c321efe687ddd31e2d0a041

4.4. Identify problem with movement and propose solution

Identify what causes the problems with movement. How can these problems be solved?

***Enemy** is currently moving 2 cells at once, what causes problems with movement. We can model the speed of the enemy differently. Instance of **Enemy** will always move 1 cell at once. However, we introduce move **delay** – instance of **Enemy** will move after **delay** calls of method **act()** will pass.*

4.5. Attribute Enemy.moveDelay

Add new attribute of type **int** called **moveDelay** into class **Enemy**. Create parametric constructor with parameter to initialize this attribute. Initialize the attribute with parameter. Adjust code in **Arena** accordingly.

Commit: 6092489ce57541e77ae4e2ee886b20853df9f8a4

4.6. Movement of enemies respecting delay

Update the method **act()** of class **Enemy**, so it moves after **moveDelay** calls of the method. Introduce new attribute **nextMoveCounter**. Initialize it to **0**. Modify **act()** method so it calls **this.move(1)** only if **nextMoveCounter** reaches **0**. After movement, reset **nextMoveCounter** to the value of **moveDelay**. If instance of **Enemy** could not move (because **nextMoveCounter** did not reach **0**), decrease **nextMoveCounter** by **1**.

Commit: bf26e6ed23911ccb712fae3e243cdedff3a89a7f

4.7. Parametric constructor of class Direction

Add parametric constructor to class **Direction** with sole parameter **rotation** of type **int**. Rotate created instance in the body of constructor by the parameter. Adjust code in **Arena** accordingly.

Commit: 3c4b9ef57ab17bac2a0abc7fc5e76ea4b6e27e4b

4.8. Overload constructors in class Direction

Overload constructors in class **Direction** by adding non-parametric constructor. In the body of non-parametric constructor, call parametric constructor with argument **direction** equal to **0**. Adjust code in **Arena** accordingly – where possible, call non-parametric version of **Direction** class constructor.

Commit: 1e67e67523c66acea4e93363c9a3173302f424c8

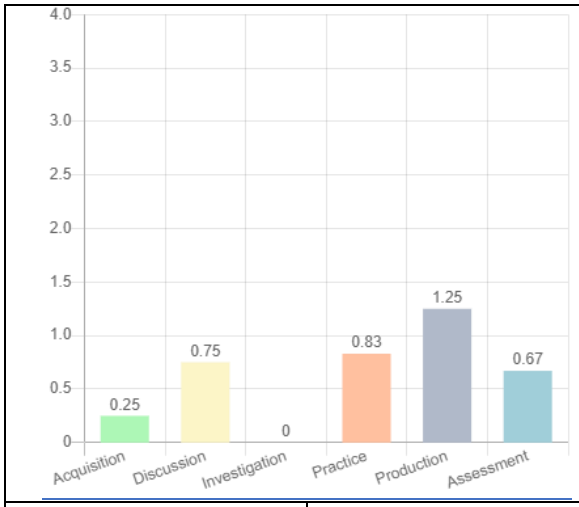
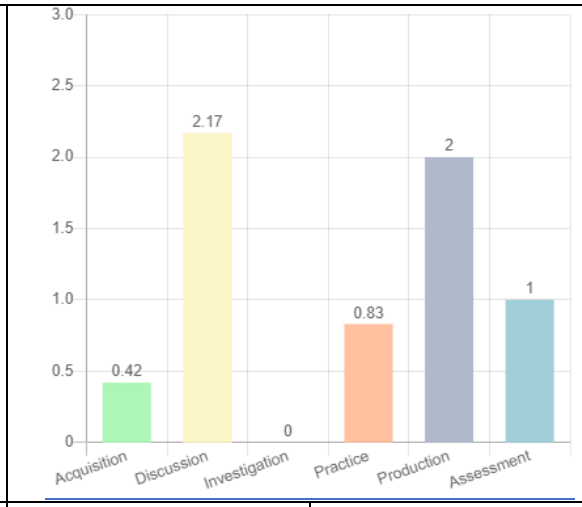
5. Association

The most important topic of this project is focused on the association. The discussion is used to find how the cooperation of different objects can bring complex behavior, even though the codes in the objects are easy to understand and maintain. UML sequence diagrams are used to illustrate the cooperation of objects and spreading the algorithm among cooperative objects. This diagram can be built during the discussion with class.

The project can be found completed after this topic. The following chapters introduce more variability to the application with focus on advantages of OOP when properly used.

Table 8 summarizes comparison of workloads of topic Association between projects Bomberman and Tower defense. We consider the understanding of association to be the most important competence when utilizing this project to teach OOP. Therefore, we significantly strengthened production and discussion TLAs. Note that there are also assessments TLAs. These are designed in a way to use previously done TLAs in slightly different context.

Table 8: Comparision of workloads of topic Association between projects Bomberman and Tower defense

	Bomberman	3h 45min	Tower defense	6h 25min
				

5.1. Discuss what should happen when enemy reaches orb

After the enemy reaches the orb, the orb decreases HP. If the HP = 0, game ends, otherwise the enemy is respawned in the arena.

5.2. Discuss how instance of class Enemy should interact with the relevant objects using messages when hitting instance of class Orb

The algorithm is spread among cooperating objects.

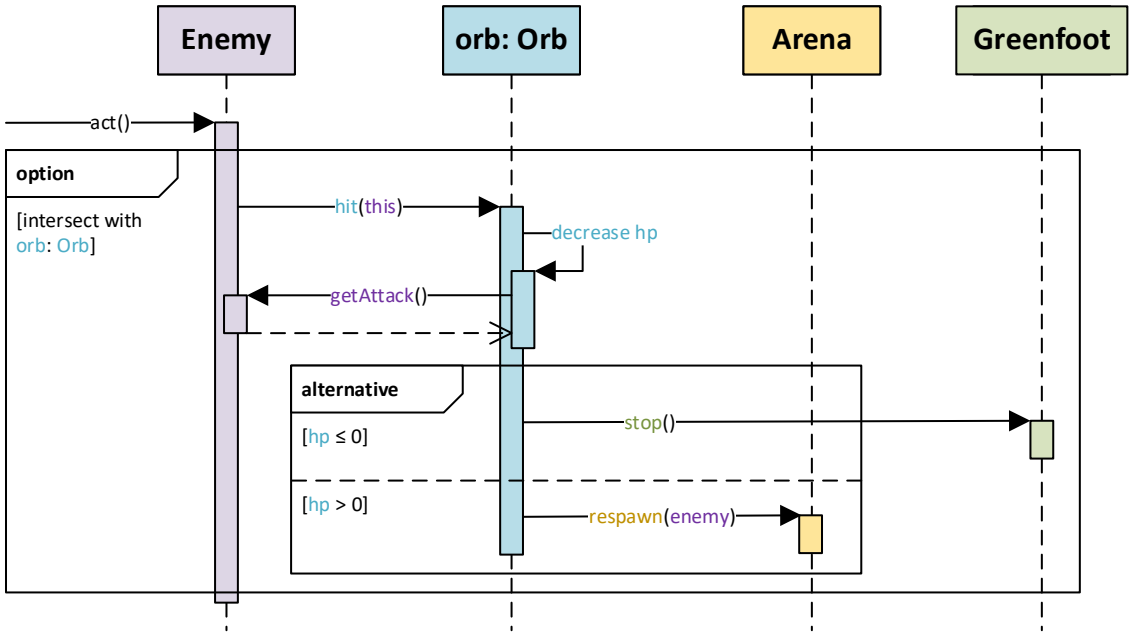


Figure 7: UML sequence diagram of instance of class Enemy interacting with other objects when hitting instance of class Orb

5.3. Attributes Enemy.attack and Orb.hp

Add new attribute of type **`int`** called **`attack`** into class **`Enemy`**. Add parameter into constructor to initialize this attribute. Initialize the attribute with parameter.

Add new attribute of type **`int`** called **`hp`** into class **`Orb`**. Add parametric constructor with parameter to initialize this attribute. Initialize the attribute with parameter.

Adjust code in **`Arena`** accordingly.

Commit: 4ca1e9f25685990d2bdfe5b610c28422e0944f95

5.4. Getter of attribute Enemy.attack

Create getter (method used to get a value of an attribute) of attribute **`attack`** in class **`Enemy`**.

Commit: 72b7456ea4cc11416c57d72c89b6a7f7e9266e3e

5.5. Create and test method Arena.respawn(Enemy)

Add method **`respawn`** without return value and with sole parameter of type **`Enemy`** into class **`Arena`**. In the method, set location and rotation of the instance of class **`Enemy`** into same values as when created in constructor.

Test the method. After the instance of **`Arena`** is created, do not launch the application. Instead drag the instance of **`Enemy`**. Then invoke context menu of the **`Arena`** instance (you need to right click in the arena where there is no instance of another class) and select **`respawn`** method item. To fill a parameter, make sure that the application is paused and filed with parameter is active (with cursor blinking inside). If so, left click on the instance of **`Enemy`**. Observe what expression has been built in the window. Then click the OK button and see what happens.

Commit: 43a221876b8acb4fd507175ec4c8f520121d1ab1

5.6. Create and test method Orb.hit(Enemy)

Add method **`hit`** without return value and with sole parameter of type **`Enemy`** into class **`Orb`**. Let the body empty.

Test the call of the method. Use similar steps as above, however, invoke the context menu of the instance of class **Orb**. Observe what expression has been built in the window.

Commit*:* fe03d520260f172066be35055a901487bf7c2ff7

5.7. Call method Orb.hit(Enemy) from Enemy

Alter code in method **act()** of class **Enemy** so the method **hit()** will be called when instance of **Enemy** (**this**) hits the instance of **Orb**.

Remove old codes that caused enemy to rotate when the orb was hit and that caused enemy to bounce from the edge of the world.

Commit*:* 63f9c96717d9d2587b60095e3b249b0158c8587b

5.8. Implement method Orb.hit(Enemy)

Implement the body of method **Orb.hit(Enemy)** with respect to the analysis made in task 5.2. Test your application.

Commit*:* 84bcd7c128faaa9313b507f7438f826ae2f47d2c

5.9. Add classes Bullet and Tower

Add classes **Bullet** and **Tower**. Use the same principles as in task 3.3.

Commit*:* ece4df70042c8f60098e14ad2cee55514897d825

5.10. Discuss how the instance of class Bullet should move and what should happen when it reaches instance of class Enemy or edge of the arena.

Bullet should move until it reaches enemy or the edge of the world. Bullet does not change the direction of the movement. The speed of the bullet can be managed using the same mechanism as in task 4.6.

5.11. Implement movement of instance of class Bullet

Apply knowledge covered in tasks 3.2, 3.4 and 4.6. Place a documentation comment into the code, where interaction with instance of class **Enemy** should happen.

Commit*:* d372827a831381b2254f838041fa4d9a42e53b82

5.12. Discuss how the instance of class Tower will shoot instance of class Bullet

*Keep in mind, that instance of **Tower** should not shoot in each call of method **act()**. Inspire by the mechanism used in 4.6. Separate relevant steps into methods of class **Tower**.*

5.13. Discuss how instance of class Tower should interact with the relevant objects using messages when shooting

Use analogic principles as in 5.2.

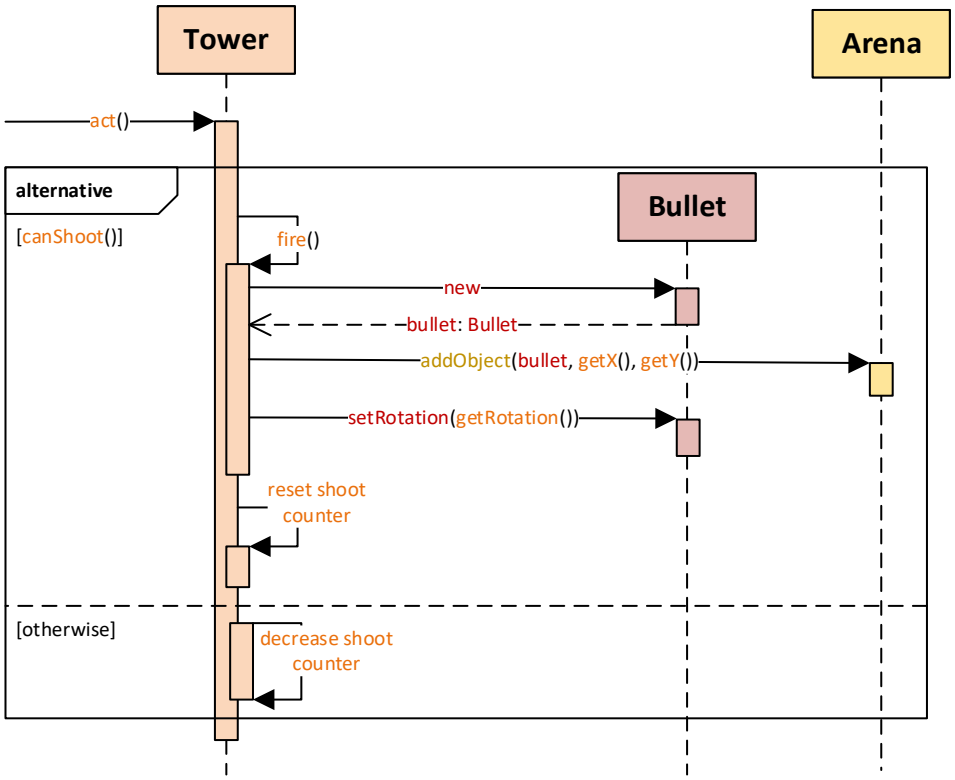


Figure 8: UML sequence diagram of instance of class Tower interacting with other objects when creating instances of class Bullet

5.14. Implement shooting of instance of class Tower

Follow output of task 5.13:

- First prepare necessary attributes and constructor, then
- create methods **`boolean Tower.canShoot()`** and **`void Tower.fire()`** (first one let **`return false`**, second let do nothing, so you can use them in method **`act()`**) and then
- implement body of method **`act()`**.

Implement method **`canShoot()`** to **`return true`** if shoot counter reaches 0.

Implement method **`fire()`** as follows:

- call constructor of class **`Bullet`** and store created instance in local variable (**`Bullet bullet`**),
- add created **`bullet`** into arena on the same coordinates as instance of **`class Tower`** (**`this`**) and
- set **`bullet`** same rotation as shooting tower.

Test your solution.

Commit: 62aec085954beacf996865a55bed312a09c675f2

5.15. Towers in Arena

Overload constructor of class **`Tower`** so it accepts parameter also **`int rotation`** (analogically to 4.8). Update constructor of class **`Arena`** to place instances of class **`Tower`** as desired. Use proper constructor of class **`Tower`**.

Commit: bfb6a271f490c341c760e654b3f86a87111c54cb

5.16. Discuss how instance of class Bullet should interact with the relevant objects using messages

Use analogic principles as in 5.2.

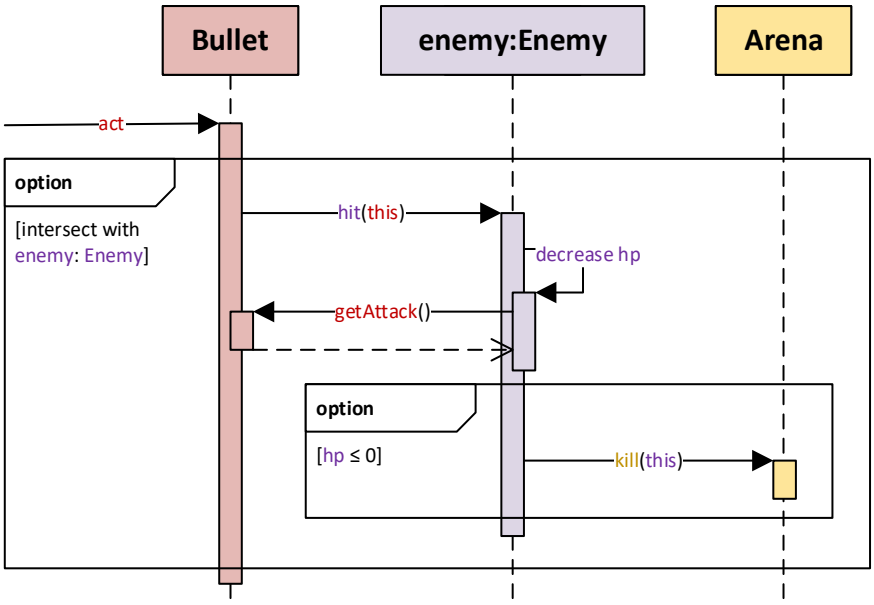


Figure 9: UML sequence diagram of instance of class Bullet interacting with other objects when hitting instance of class Enemy

5.17. Implement instance of class Bullet hitting instance of class Enemy

Follow output of task 5.17:

- first prepare attribute and methods (analogically to tasks 5.3, 5.4, 5.5 and 5.6),
- then call method **`Enemy.hit(Bullet)`** from instance of class **`Bullet`** (analogically to 5.7) where the comment was left from 5.11 and,
- lastly, implement method **`Enemy.hit(Bullet)`** (analogically to 5.8).

Test your solution.

Commit*:* dcfe31bc006b7f3dcd8b8b759cc1be901c32913c

5.18. Spawn of enemies and end of the game

Use method **`Arena.act()`** to call spawning of enemies. Properly implement delay between enemies spawn. Spawning process (create instance of **`class Enemy`**, assign its properties, add it into arena) implement in **`Arena.spawn()`** method. Count created instances of **`class Enemy`** in attribute of class **`Arena`** (initialized to 0, increase when spawned, decrease when killed). Alter method **`Arena.kill(Enemy)`** – if last enemy is killed, player won the game – stop Greenfoot and write a message on the screen.

Commit*:* d48341a095561500af6032d5c8f56e201060f9a4

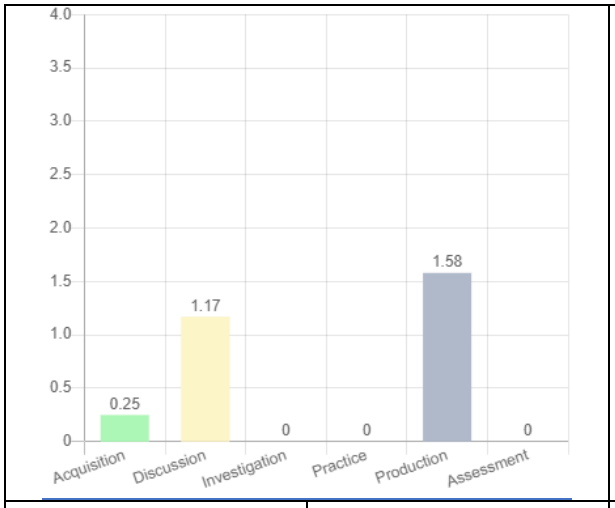
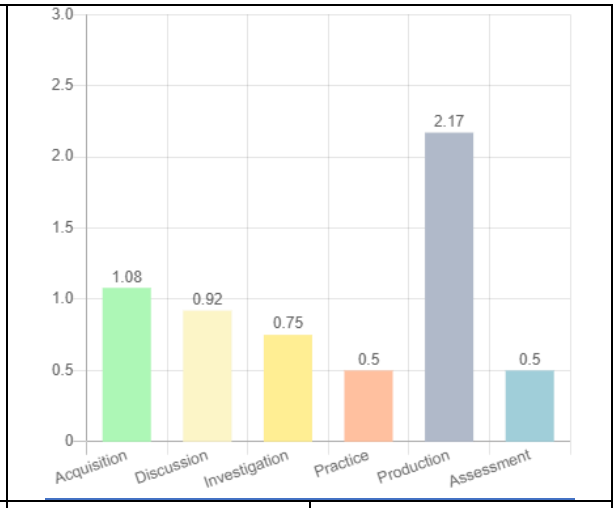
6. Inheritance

This topic introduces variability into the project via inheritance. We introduce Liskov substitution principle to show benefits of OOP. We strongly suggest to let students experiment and come up with custom subclasses of enemies and arenas. Since these will have a common interface, it will be easy to put everything together. Similarly to topic 4 there may be many home assignments created.

As mentioned before, the project can be found to be completed after previous topic. Therefore if needed, teacher may adapt this topic to show benefits of inheritance and with it connected universality only on here proposed hierarchies of class **Arena** or **Enemy**. This will lead to reduction of hours associated with this topic.

Table 9 summarizes comparison of workloads of topic Inheritance between projects Bomberman and Tower defense. The suggestion in experimenting is projected in more investigation, practice and production type TLAs. More theory is covered in this topic with focus on Liskov substitution principle.

Table 9: Comparision of workloads of topic Inheritance between projects Bomberman and Tower defense

	Bomberman	3h 0min		Tower defense	5h 55min

6.1. Identify common properties of classes Orb and Direction

*Instances of classes Orb and Direction do not act during lifetime. They just react to messages. We can introduce common ancestor that will keep method **act()** empty and make subclasses more transparent.*

6.2. Add class PassiveActor as ancestor of classes Orb and Direction

Create new class **PassiveActor**. Alter codes of classes **Orb** and **Direction** to be descendant of **PassiveActor**. Remove method **act()** from classes **Orb** and **Direction** – it is inherited from **PassiveActor**.

Commit*:* afe617814c07a5d885ed06479bf71deda8725f19

6.3. Make class PassiveActor abstract

When creating methods of the **PassiveActor** class, we encounter methods that cannot be implemented in the common class and so we have to leave the implementation to the descendants. As an example, let's consider the **Shape** class with the children **Rectangle** and **Triangle**. Each shape will have perimeter and content methods implemented, but they cannot be implemented in a common class. If we mark a class method as abstract, we are essentially saying that a descendant will implement it. The class containing the abstract method must be abstract. Therefore, we add the word **abstract** to the class header.

Commit*:* f7a5702cae29bf21c9c88620d01ef64e4127c21c

6.4. Identify common properties of classes Bullet and Enemy

Instances of classes Bullet and Enemy act similarly during lifetime. They move the same way and afterwards they react to the surroundings. We can introduce common ancestor, that will implement method ***`act()`*** *to move in same way and make subclasses focus on their specific purpose.*

6.5. Add abstract class MovingActor as ancestor of classes Bullet and Enemy

Use similar approach as in 6.2.

Commit*:* 43e53b533563ce0a860b294ad9009f77409c48d4

6.6. Identify attributes of classes Bullet and Enemy required for movement

Investigate method **`act()`** of respective classes. Identify attributes **`moveDelay`** and **`nextMoveCounter`**. Observe, that code of method **`act()`** responsible for movement is the same.

6.7. Move code responsible for movement into class MovingActor

- Move attributes identified in 6.6 from subclasses **`Bullet`** and **`Enemy`** into **`MovingActor`** (remove them from subclasses).
- Add parametric constructor into class **`MovingActor`** to initialize these attributes.
- Call parent constructor with proper parameters from subclasses **`Bullet`** and **`Enemy`**.
- Move code responsible for movement in method **`act()`** of subclasses **`Bullet`** and **`Enemy`** into **`MovingActor`** (remove code from subclasses, keep there the rest of the method).
- Call parent version of method **`act()`** as first line of method **`act()`** in subclasses **`Bullet`** and **`Enemy`**.

Commit*:* ca1f010a63445c1847b74259a1c6cd4817121db3

6.8. Create custom enemies

- Add subclasses of class **`Enemy`** that will represent different enemies (e.g. **`Frog`** and **`Spider`**). Make sure that images do not exceed the size of cell.
- Add to classes parameterless constructor, that will call parent (**`Enemy`**'s) constructor with parameters specific for each kind of enemy.
- Remove method **`act()`** (or add call **`super`**).

Commit*:* b0ac1fbe793548a32f7700c292aed631918c8388

6.9. Spawn custom enemies

Update method **`Arena.spawn()`**. Create instance of either **`Frog`** or **`Spider`** and store it into variable of type **`Enemy`**. Use any kind of decision to decide which instance should be created (may be random, may be precisely counted, etc.). See that no other codes in the application must be changed.

Commit*:* 8cd4397f585ec957bbc18ca98e01823f434a13a6

6.10. Discuss hierarchy of Arenas

Discuss and design hierarchy of **`Arena`** classes. Subclasses of **`Arena`** will be responsible for custom layout – position of **`Orb`** and **`Direction`** instances, size of arena. These tasks will be performed in constructor of subclass. What will be needed to pass into parent class (**`Arena`**) constructor parameters? Keep in mind that all the rest (spawning, respawning and killing enemies) will be performed in class **`Arena`**.

You should identify the need to set and store spawning position and rotation. This will be done using attributes and relevant constructor parameters. Moreover, constructor should also accept dimensions of area.

6.11. Make universal Arena

Introduce attributes **`int spawnPositionX`** , **`int spawnPositionY`** and **`int spawnRotation`** and use them in methods **`spawn()`** and **`respawn(Enemy)`**. Add parameters to constructor of class **`Arena`** to initialize them.

Add two more parameters to constructor of class **`Arena`** – **`int width`** and **`int height`**. Pass these parameters to parent constructor.

Notice that **`Arena`** cannot be automatically constructed by **`Greenfoot`**, since it needs parameters for constructor. Make it **`abstract`**.

Commit*:* e9844d7d9b5f19969618b469ebc907d0fe3c1357

6.12. Create DemoArena

Add subclass **`DemoArena`** of class **`Arena`**. Call parent constructor of class **`DemoArena`** with parameters that will ensure to create arena of same dimensions and spawning enemies in a same way as before.

Move code responsible for layout of instances of classes **`Direction`**, **`Orb`** and **`Tower`** from constructor of **`Arena`** to constructor of **`DemoArena`**.

Create instance of **`DemoArena`** – from context menu of **`class DemoArena`** select item **`new DemoArena()`**.

Commit*:* 6a6569774b5735f453a56c7cb2cdbf19d228eae9

6.13. Create custom arenas

Using similar approach as in 6.12 create other innovative subclasses of class **`Arena`**. You can share the code with other students in your group.

7. Encapsulation

The last topic is focused on proper utilization of private methods and class methods and variables. The usage of class methods and variables can be replaced by (non-class) attributes and methods in class **`Arena`** (in the context of our project there is only one instance of **`Arena`** present). This will allow to implement tasks from this topic but using the already known concepts.

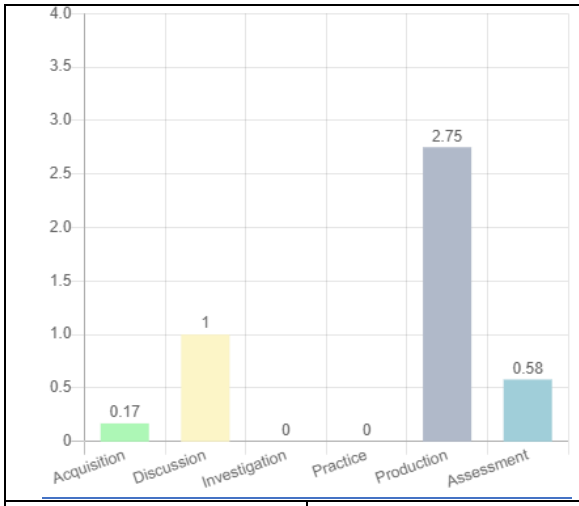
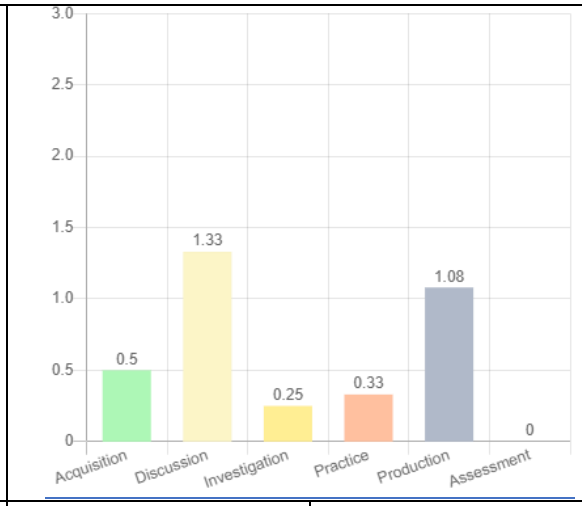
We have already used encapsulation for the Bomberman project. Here, static attributes and static methods will be used. Until now, all attributes and methods were bound to an instance of the class. Let's imagine a situation where we would like to count bullets fired in the **`Bullet`** class. The number of bullets fired is common to all instances of the **`Bullet`** class and does not depend on a particular instance. Such an attribute is called static and since it is common to the class, it is accessed through the class name, e.g. **`Bullet.count`**. Similarly, we can create, for example, a method that returns the number of bullets. Again, this will be common to all instances of the class. It will therefore be static. Static attributes and static methods have the keyword **`static`** in their declaration. Note that they can exist even if there is no instance of the class. Static methods are initialized in the definition itself.

```
static int countOfBullets = 0;

static int countOfBullets(){

...
```

Table 10 summarizes comparison of workloads of topic Encapsulation between projects Bomberman and Tower defense. Similarly to previous topic, the production is more evenly distributed among other types of TLAs. The higher amount of acquisition TLAs comes from the introduction of class methods

and variables. As suggested, there is a possibility to avoid these concepts, what will lead to reduction of those TLAs, what will lead to similar design as of project Bomberman.

Table 10: Comparision of workloads of topic Encapsulation between projects Bomberman and Tower defense

	Bomberman		Tower defense
4h 30min		3h 30min	

7.1. Create class ManualTower

Create **`class ManualTower`** as descendant of **`class Tower`**. Prepare images of this class when under control and when not. Add two constructors with same signature as the parent constructors and ensure calling of parent constructor. Let method **`act()`** call parent version of itself.

Add instances of this class into the layout of chosen **`Arena`**.

Commit*:* 63a02fa0c5080165cba8b467da08c4b65f31d0a8

7.2. Change of control of manually controlled tower

Add attribute **`boolean isManuallyControlled`** and initialize it to **`false`**. Create method **`void changeControl(boolean)`** and change the attribute and change the image accordingly.

Commit*:* 2257746b7dac5eaab7acc55d6493319230338f3a

7.3. Invoke change of manual control

Manually invoke change of manual control of selected instance of **`class ManualTower`**. Observe changes in internal state similarly to 1.6.

7.4. Process user control

Create private method **`void processUserControl()`**. First detect, if it was clicked on this instance. If so, change to manual control. Afterwards implement manual control itself. Test, whether the instance is in manual mode and if so, obtain **`MouseInfo`** object. If the object was obtained, turn the instance of **`ManualTower`** towards the position of the mouse cursor.

Call prepared method **`processUserControl()`** from method **`act()`** before passing the execution to parent (**`super.act()`**). Check in the method, whether it was clicked on this instance. If so, call method **`changeControl`** with proper value.

Test your solution by executing 7.3 again.

Commit*:* 6ec1f489576019a6493490f9e97797920b923869

7.5. Identify problem with user control and propose solution

Identify what is problematic with user control. How can these problems be solved?

Currently it is not possible to deselect tower. There should be an evidence to currently controlled instance, that will be deactivated when some other instance will be selected. Add evidence of manually controlled tower.

7.6. Add evidence of manually controlled tower

Add attribute into class **`ManualTower`** of type **`ManualTower`** to represent reference to manually controlled instance and initialize it to **`null`**. This attribute must be common to all instances of the **`ManualTower`** class, so it must be defined with the **`static`** keyword. Inspect internal state of class (from context menu of **`class ManualTower`** select menu item **`Inspect`**). What was added?

Commit*:* c4739460bed583d2126de066acc6b1149d022990

7.7. Change of manually controlled tower from centralized place

Add method **`changeControlledInstance`** to change manually controlled tower as class method of class **`ManualTower`** (thus defined with the keyword **`static`**). Parameter of the method should be reference to instance of **`ManualTower`** that will be manually controlled.

If the method parameter differs from the manually controlled instance (in a static attribute), use the **`changeControll`** method with the correct parameters. First, use **`changeControll`** to set the original manually controlled instance to **`false`**. Then change the static attribute of the **`ManualTower`** class of the manually controlled instance to the parameter of the new method (i.e., the new manually controlled tower). Finally, use the **`changeControll`** method to set the new manually controlled instance to **`true`**. Don't forget to handle possible **`null`** references. This is because there could be a situation where there is no manually controlled tower.

Test your solution. From context menu of **`class Tower`** select menu item with newly created method. To fill a parameter you can use the same principle as in 5.5.

*You should observe, that newly created class method is not consistently called, Instances of **`ManualTower`** bypass the evidence when processing input, what causes problems.*

Commit*:* 9dc6d8dd4dcbbd71edb8009c1a72403dea1a0ee0

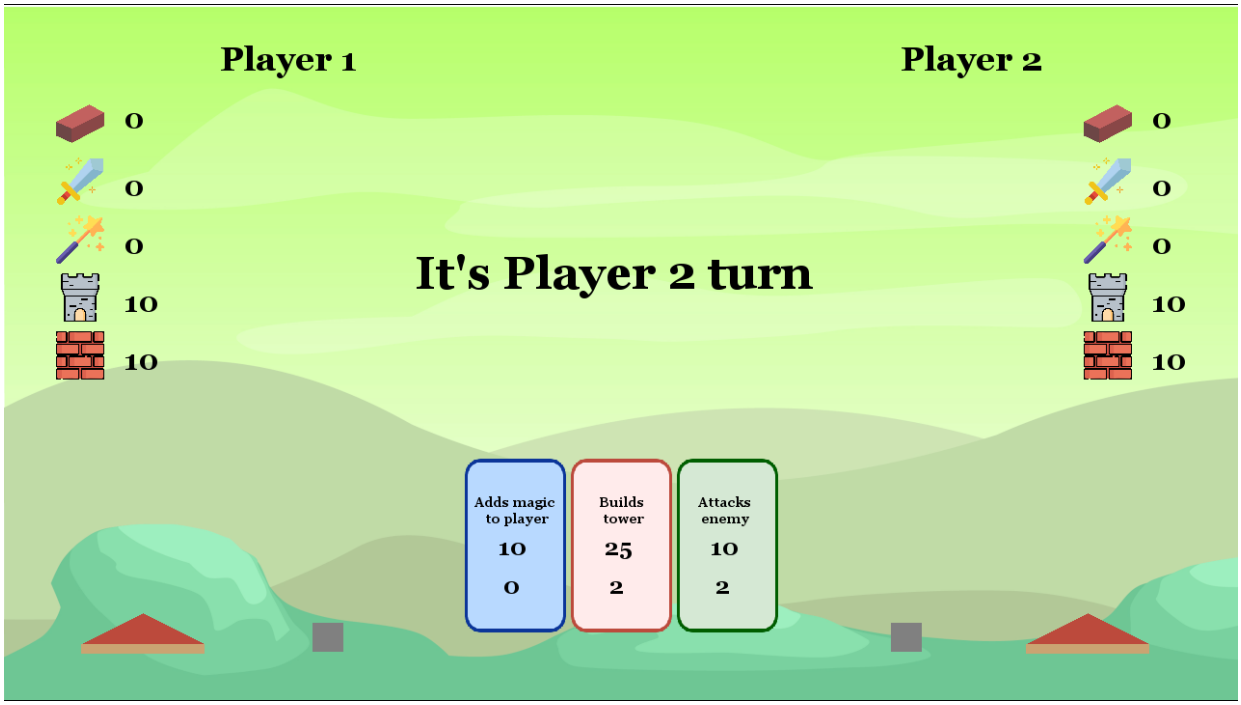
7.8. Invoke change of manually controlled tower

Invoke **`ManualTower.changeControlledInstance(ManualTower)`** from relevant places. Lastly make method **`ManualTower.changeControl(boolean)`** private. Observe changes in interface of instance of **`class ManualTower`** similarly to 1.6.

Commit*:* c052bbb6aa4c7e690d4d8cf55d3831028fa2b9e3

3.3. Project ants

Ants is a card-based game for two players. Each player has its own tower and wall and resources such as bricks, swords and magic. One turn of a game consists of an action, where the game offers the player 3 cards at random, and he chooses one. There are three types of cards – building cards, fighting cards and magic cards. Building cards can be used to increase your own tower or wall, fighting cards to attack an enemy player and magic cards to increase the number of own resources or steal enemy's one.



Source codes are available at:
https://gitlab.kicon.fri.uniza.sk/oop4fun/project-ants

Learning design is available at:
http://learning-design.eu/en/preview/67aa1d089763d07f29809d42/details

3.3.1. Topics

Project Ants is divided into 6 topics:

Covered topics of light OOP are:

- classes, objects, instance
- methods, passing methods arguments
- constructors
- attributes
- static variables and methods
- encapsulation

1. Greenfoot environment, Class definition, basic work with classes

The topic is devoted to project creation. Students will be capable to create new project in Greenfoot environment, create class (as subclass of **`Actor`**), select background image, create instance of create class and send it a message.

Create a new project. Give it a proper name (e.g. **`Ants`**) and save it to a proper location.

Table 11 summarizes comparison of workloads of first topic between projects Bomberman and Ants. The total workload of both projects is the same in this part.

Table 11: Comparision of workloads of topic 1 between projects Bomberman and Ants

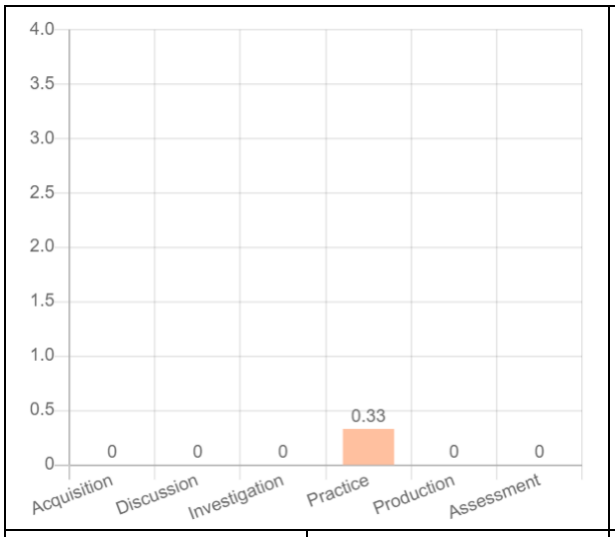
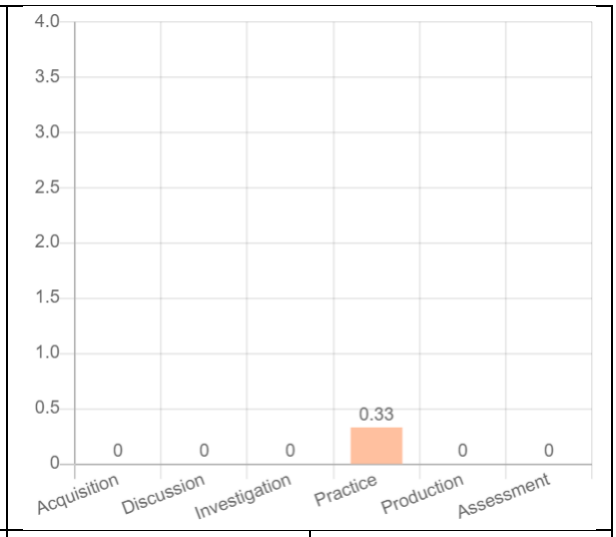
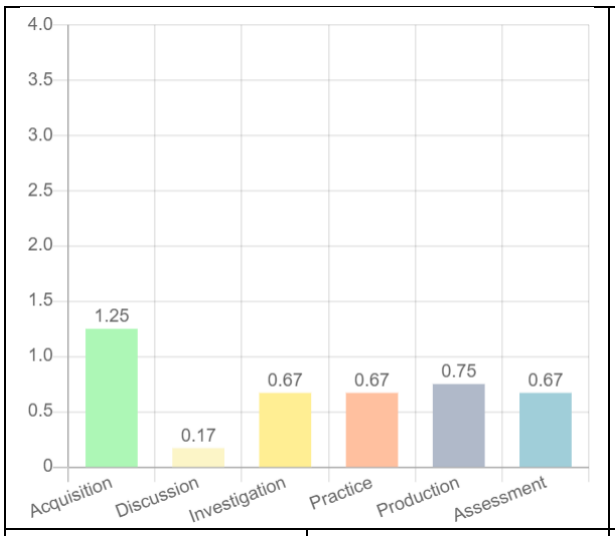
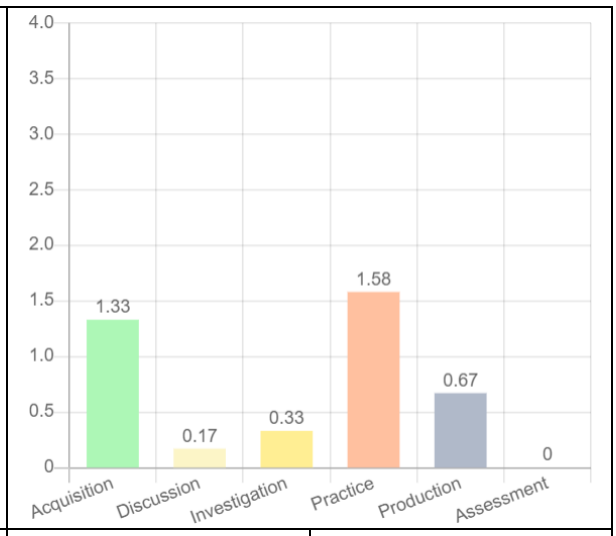
			
Bomberman	0h 20min	Ants	0h 20min

Table 12 summarizes comparison of workloads of topic Class definition and basic work with classes between projects Bomberman and Ants. The total workload of both projects is similar. The main difference is in practice and assessment. However, as will be provided in the next section, some of the practice tasks can be also given as assessments, what can balance it even more.

Table 12: Comparision of workloads of topic Class definition and basic work with classes between projects Bomberman and Ants

			
Bomberman	4h 10min	Ants	4h 05min

1.1. Introduction to Greenfoot

Create a new project in Greenfoot and introduce students with basic elements, UI etc. Initial state of repository contains also assets you can use in project.

Commit*: 2f0658d99a7abcbad6399f63c369dcd4c053af2a*

1.2. Creation of class Wall

Create class **`Wall`** as a child of **`Actor`**. Introduce concepts such as classes, class hierarchy, instances etc. to students.

Commit*: cd163c8a3b1c952760a3a9e24ec6a322939a8ea6*

1.3. Creation of class Tower

Similarly, as previous task, create class **`Tower`**. You can leave it to students as an individual assignment.

Commit: a0c1405183704f61156732c9bd55cdc921d95adc

1.4. Defining class attribute/field

Introduce terms field/attribute, primitive types, etc. to students. Try to identify fields in our classes (lead students specifically to **`height`**) and define it in class **`Wall`**.

Commit*: b6bd179478e1440249080d4cdcc00b4428bef080*

1.5. Assigning value to attribute/field

Talk about field values and assignments and assign to wall **`height`** value 10.

Commit*: 917861df37d13c09d840008e0c7f7d263ea56c95*

1.6. Defining and assigning value to attribute/field for class Tower

As an individual work leave students to repeat the same for class **`Tower`**.

1.7. Class constructors

Talk about class instantiating and constructors. Move assignment of value to constructor.

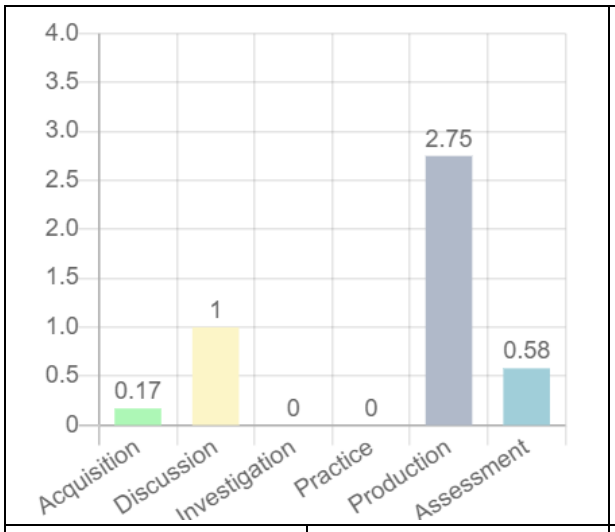
Commit*: a85930c312f079d65137eba8e147a8ba2b99e84f*

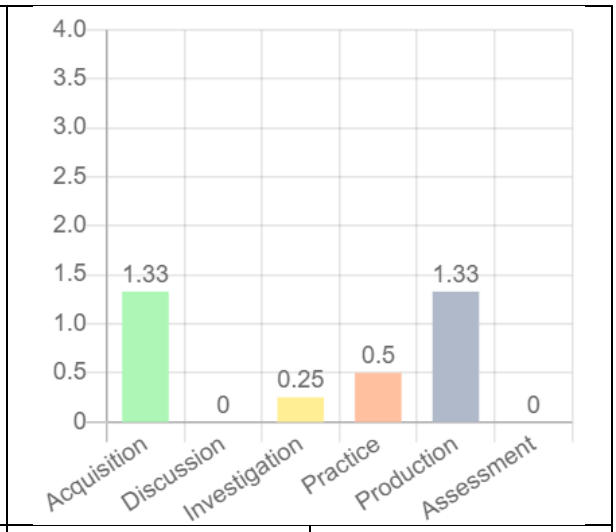
2. Encapsulation, composition, methods

In this section, the basic principles of OOP are provided, specifically terms like encapsulation, composition and methods. Students will learn how and why fields/attributes should be encapsulated and not provided as public, how objects are composed with other objects and how to create methods and call them. Students will create an object of **`Player`** in this section and give it fields of object type – **`Wall`** and **`Tower`**.

Table 13 shows differences in two similar topics – Encapsulation from Bomberman project and Encapsulation, composition and methods in Ants. As can be seen, Ants project gives more focus on acquisition and on the other hand, Bomberman focuses more on production and discussion. This is caused by the fact that this topic in ants consists of more than encapsulation, therefore also more acquisition is required. The total workload is an hour shorter in Ants than in Bomberman as these concepts are explained more shallowly but in the next sections, you would consolidate this knowledge.

Table 13: Comparision of workloads of topic Encapsulation, composition, methods between projects Ants and similar topic in Bomberman - Encapsulation





Bomberman	4h 30min	Ants	3h 25min

2.1. Defining methods

Talk about encapsulation, explain to students why it is not good practice to make for example wall **`height`** as public property and rather encapsulate it in getter. Then create method **`getHeight()`** in **`Wall`**.

Commit*: 33af3b2d18a7b6c759c16b1bb0a4632f648a4d85*

2.2. Defining methods with parameters

Explain method parameters and define method **`increaseHeight`** in wall, that will increment wall **`height`** by specified number.

Commit*: 50cfefc2747ee3150a16f66f439a9391fbff922a*

2.3. Repetition for class Tower

As an individual work you can give students the task to repeat it also for class **`Tower`**. This task doesn't have an associated commit as the work is quite simple. In the next commit, there are also changes for this task, so you can compare it if you are not sure about the result.

2.4. Object composition

Explain to students what composition is and why it is necessary to have object types as fields. Try to identify such types for each player in this game. Then create object **`Player`** and give it fields/attributes **`Wall`** and **`Tower`**.

Commit*: 50b6b825952c02f9743e08bd1bb8415aa6f08eef*

2.5. Creating instance of class

Explain constructors and create **`Wall`** and **`Tower`** instances in the **`Player`** constructor.

Commit*: 873174a2851f5b5e6054f8071412058fc97e6e2e*

2.6. Calling methods of instance

Explain to students how you can call methods of created instances. Then try to encapsulate them so one is able to call it from outside of **`Player`** through player instance. Create methods **`getWallHeight`**, **`getTowerHeight`**, and **`increaseWallHeight`**, **`increaseTowerHeight`** in **`Player`** object.

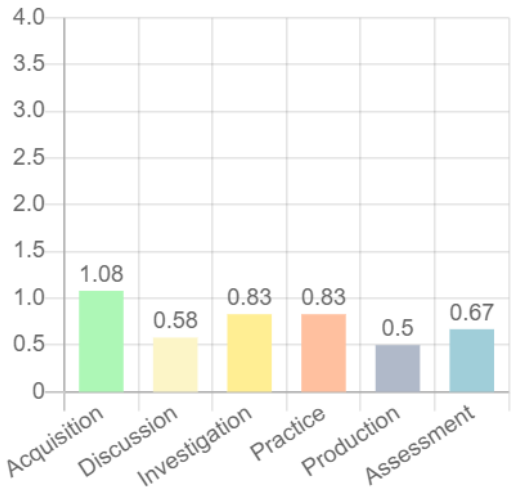
Commit*: b787d7f70ee6772b0185d5265f8ff56191a775e9*

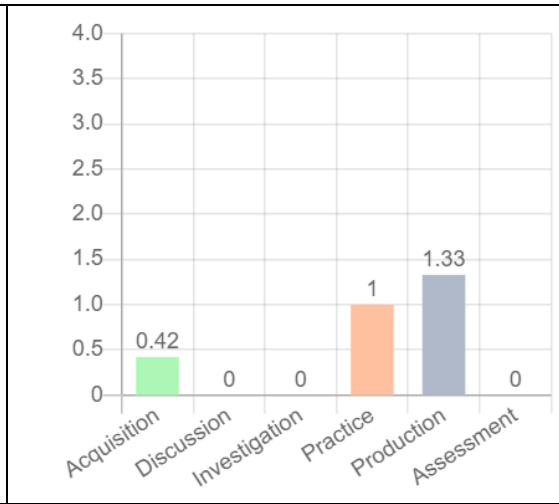
3. Constructors, more complex method calls (working with graphic in Greenfoot)

This section is focused on methods calls using calling of Greenfoot objects for drawing our instances. Students will draw instances of wall, tower and player.

Table 14 shows comparison of this topic in project Ants and the most similar topic in project Bomberman. Please note that these topics are a bit different, as ours shows basic work with constructors and introduces graphics in Greenfoot and in project Bomberman it is more focused on algorithm. Therefore, also the differences can be pointed out because of this.

Table 14: Comparision of workloads of topic Constructors, more complex method calls (working with graphic in Greenfoot) between projects Ants and similar topic in project Bomberman, that is covered in topic Algorithm





Bomberman	4h 30min	Ants	2h 45min

3.1. Drawing objects in Greenfoot – Wall

Introduce constants in Java to students – define **`wallSizeX`** and **`wallSizeY`** as a static final (the **`final`** keyword ensures that the value cannot be changed, i.e. that it is a constant) attributes (constants) with values 32 and 3. The constant is accessed as a static variable via the class name, e.g. **`Wall.wallSizeX`**. Then implement function draw in **`Wall`**, where new image is created with given size and filled as rectangle.

Commit*: 3d728f73182b79b36d95a5bb1742cd870d82f906*

3.2. Drawing objects in Greenfoot – Tower

The same is repeated for **`Tower`**, however it is more complicated as **`Tower`** consists also of roof what is implemented as a polygon. Polygon requires an array of points. If you want, you can give students a quick explanation, although arrays are not part of this section. Have a conversation with students about the analogy of the terms array and list.

Commit*: 98e55ae90a47fffd79152eabd80b6e13a15d91d5*

3.3. Defining other properties of Player

Try to identify other properties of the **`Player`** – specifically the number of bricks, swords and magic and the name of the **`Player`**. Then implement these properties in the **`Player`** and initialize them in constructor.

Commit*: 6d2b7771e8abe90117d61676215a39592ed41d39*

3.4. Drawing Player

The last task in this section is drawing of a **`Player`**. You have to draw icons of each resource and the number of such resource and also player's name.

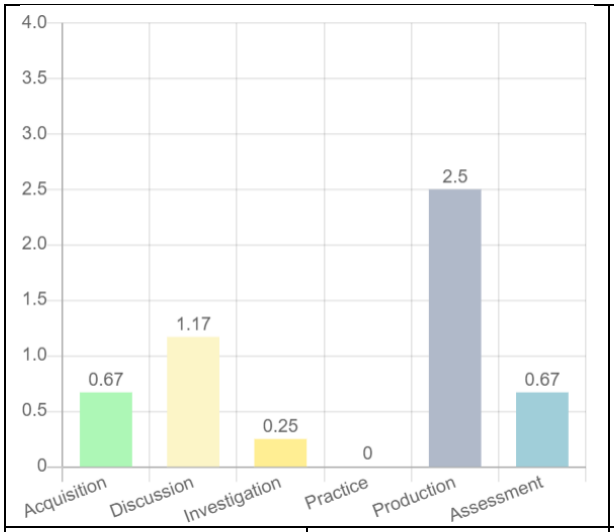
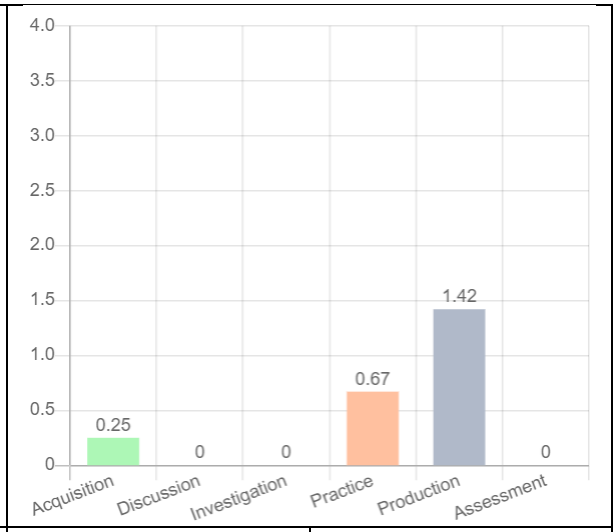
Commit: *a07482609d4494babdab16e67e221523d8cb5683*

4. Branching, conditional execution

This section focuses on branching of program and conditional execution of parts of our game. There are several cases where this is necessary in this game, such as drawing the first player in the left part of screen and the second in the right etc.

Table 15 shows differences between similar topic in Bomberman and Ants. As can be seen, Bomberman project gives greater emphasis on discussion as project Ants. Also, the total workload in hours is approximately half that in Bomberman. This is caused by the fact that branching is split into multiple sections in this project – this section is more of introduction and basic usage of branching.

Table 15: Comparision of workloads of topic Branching, conditional execution between projects Bomberman and Ants

	Bomberman	5h 15min		Ants	2h 20min

4.1. Creating class Game

Before we start to put branching logic in our code, we need to create an object **`Game`** that will hold both players and, in the future, will manage game logic, turn switching, card execution etc. For now, we will put there properties for two players and create their instances in the **`Game`** class constructor.

Commit: *06aaf814f2adcc0cda1dd20f5993eea3fcfcbf2e*

4.2. Branching, conditioning execution of code – players are shown on corresponding sides of the game plan

Now we should introduce a new attribute for our **`Player`** – the information whether the player should be plotted on the left or right side of the screen. We will use this information to set corresponding properties of player – position of **`Wall`**, **`Tower`**, "hud" and **`name`**. This offset should be then added to the **`redraw`** method.

Commit: *c84a9065d8a5c251b2f25c61720ed8e54eb5f1d7*

4.3. Adding instances to world

The next task creates an initial view of the **`Player`** class instances for the game and instantiates the game in the world.

Commit*: f6102678b62423d9888f7beea802129d550e19cd*

4.4. Adding instances to world 2

For the **`Player`** to be correctly drawn in the **`Game`**, we need to add **`Wall`** and **`Tower`** to the world and call **`redraw`** function in **`act`** method. Here you can explain execution of game loop (**`act`** method).

Commit*: eb53be86180b4f3e043ea5b0363fccef559d4c58*

4.5. Conditioning execution code only one time

When you try to run the **`Game`** after the last task, you can encounter a problem – objects are added to the world multiple times in a second. You can give students assignments to fix this problem or fix it with them. One of the solutions is to introduce a new boolean attribute, that would store information if the initial object added to the world was executed or not and after first execution of act method this property is set to true.

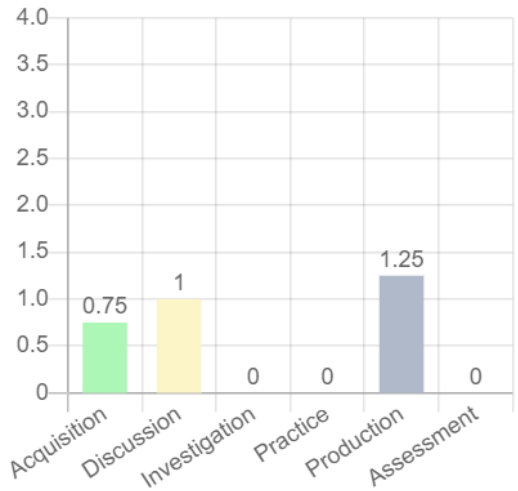
Commit*: 707304650909d8450bac27a5264cdaef0e103c6a*

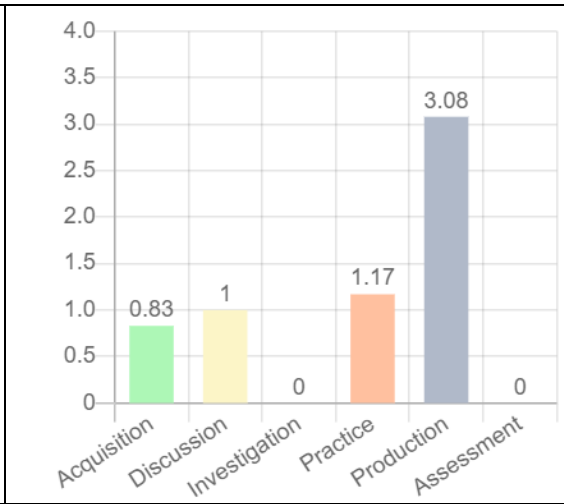
5. Algorithm, enumerations, arrays

This section discusses the next concepts, like algorithm, enumerations, arrays and loop over array of elements. In this section, students will implement cards that will be used to play the game.

Table 16 contains comparison between similar topics in projects Ants and Bomberman. As can be seen, project Ants gives more focus on production and practice and slightly more on acquisition. Also, the total workload is about two times bigger than in the project Bomberman. This is caused by the fact that this topic consists of more than working with lists, but also introduces concepts like enumerations etc.

Table 16: Comparision of workloads of topic Algorithm, enumerations, arrays in project Ants and similar topic in projects Bomberman - Lists





Bomberman	3h 00min	Ants	6h 05min

5.1. Implementing card class

Firstly, we can discuss with students how the final game would work, and design **Card** object. You should come to a solution that will hold information about card type, its requirements, effect and some description. Then you can create such object as a child of **Actor**.

Commit: *928eabaebc330c865d9eb1e28d1a03453c3031ba*

5.2. Enums

In the previous task, you created card type as a field in **Card** object. Discuss with students what type this field should be – **String**, **int**, etc. and you can introduce enums to them. **Enum** is a data type with a finite set of named values (e.g. for days of the week it is the values: Monday, Tuesday, Wednesday, Thursday, Friday Saturday and Sunday). Create **CardType** enum with them and specify its individual values.

Commit: *d983729a3c57d73ef2e028e39d7f067a725ba1d0*

5.3. Branching – switch

Now we should implement drawing of a **Card**. This is based on the card type, to visually divide cards. You can show students how this would be done using **if** and compare it with **switch**. There are three card types – building cards, attacking cards and magic cards. Based on this category, background is selected. In case we will do it using **if** statement, the code should look like this:

```
if(type == BuildTower || type == BuildWall || type == IncreaseBricks) {
        background = new GreenfootImage("building-card.png");
} else if(type == IncreaseSwords || type == Attack)
...
```

This can be replaced by **switch** statement. Java switch statement works in such a way, that specific branch will be executed, however it won't stop executing other branches unless you call **break** statement. In our case this allows us to merge multiple branches together and write our assignment statement only for the last card type of such category. Therefore, this code:

```
switch (type) {
        case BuildTower:
                background = new GreenfootImage("building-card.png");
                break;
        case BuildWall:
                background = new GreenfootImage("building-card.png");
                break;
        case IncreaseBricks:
                background = new GreenfootImage("building-card.png");
                break
        ...
}
```

can be written also in a way we use in this taks:

```
switch (type) {
	case BuildTower:
	case BuildWall:
	case IncreaseBricks:
		background = new GreenfootImage("building-card.png");
		break
	...
}
```

Commit*: 48e9df2d8eb83d6a88d0c86667c9b43497f065f2*

5.4. Array

Game object should hold **Cards**. This can be done by introducing three fields of **Card** type (you can also extend hand – i.e. the number of **Cards** **Game** offers to **Player** in one turn – to more of them). You can explain to them that it would be impossible to store even more cards, when we decide to extend hand even more and you can explain the concept of arrays to students. You should also create an instance of an array.

Commit*: 2261666afe20ba205d252c8540ff6aa22177751b*

5.5. Simplifying instantiating of cards – CardFactory

When creating new **Cards**, a lot of information should be provided. You can try to find a solution to this problem. One of the solutions is to create a **CardFactory** that will hold instances for each card and implement **clone** method on **Card**. Therefore, new **Cards** can be created using this **CardFactory** and cloning existing ones.

Commit*: c3135c874c2c1181b4448e338c977ed1c9fba317*

5.6. Random – Instantiating of random card

After implementing **CardFactory**, also **clone** method should be implemented, as were discussed before. The **CardFactory** should also be able to give instances of random **Card**. You can explain random number generator and implement method, that will **clone** random card and also random base card (base cards are cards with no cost - this is implemented so as we can guarantee **Player** can always play at least one of provided **Cards**).

Commit*: 462f2a8583b37636d4a9c6fff2ddd09520485d51*

5.7. Loop over array

Now we have to create an instance of the **CardFactory** in the game and implement generating of **Cards** and clearing them (removing them from world) - when explaining this, you can introduce some form of loop.

Commit*: fd7dae3ec277ef66986f63ae8c85481c4c9f22d3*

5.8. Drawing of Game

The last task in this section is to implement the drawing of the whole **Game**. During this, we should also prepare our **Cards** using a previously created method and also introduce information if the first player is active. Drawing of a **Game** should consist of drawing of all **Cards**, and drawing information what player is currently on turn.

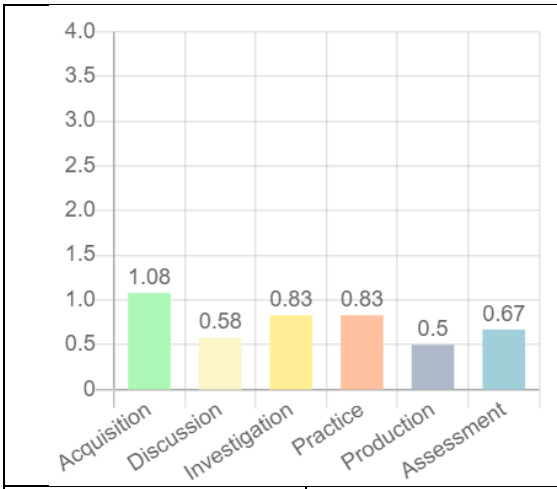
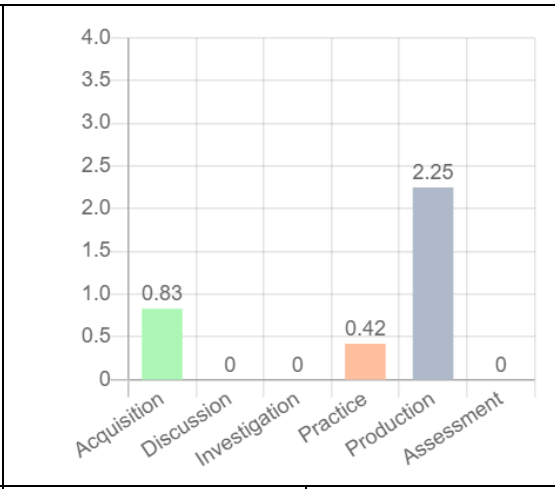
Commit: *f7f6b68566ebe2a1c66bc284c3542e562489100b*

6. Handling user input, Game logic

This section focuses on handling user input – how to get, for example, the players name from user, how to handle clicking on cards and how to finish game logic. In this section also some advanced concepts are introduced (singleton).

Table 17 shows comparison between this topic and the most similar topic in the project Bomberman – Algorithm. Please note that as this is the last topic of this project, many concepts introduced in the project Bomberman are already known to students at this point of this project. Therefore, there is markedly less focus on investigation and discussion than in the project Bomberman and a lot more production.

Table 17: Comparision of workloads of topic Handling user input, Game logic in project Ants and similar topic in project Bomberman - Algorithm

			
Bomberman	4h 10min	Ants	3h 30min

6.1. Input names from user

In this task, you should explain dialog windows (Greenfoot ask method) to students and set **`Player`** names according to these inputs.

Commit: *17b9237fdabb19bb16d7c276fa06ce242dac3404*

6.2. Static instance of class – Game as a singleton

`Game` objects should be constructed exactly one time – you can discuss this problem with students, and you can provide a solution in the form of singleton – static instance of **`Game`** object and **`private`** constructor. Singelton is used in cases when we want only one instance of a certain class to exist in the whole application. A well-known example of a singleton is e.g. the Package in the operating system). This is required because when a **`Player`** uses a card, there should be a reference to the **`Game`** so that clicking on it can be handled correctly, as will be discussed in the next part. The other way to implement it (without singleton) is to provide instance of **`Game`** to the constructor of the **`Card`** and the **`CardFactory`**.

Commit: *fd4369bd7b88f15119354d7f243c9093c769a964*

6.3. Handle input click on card

Mouse click is handled by calling **`Greenfoot.mouseClicked`** method in act. When clicked, you should call method **`useCard`** of **`Game`** and send reference to self (**`this`**).

Commit: *d1846818d19011202ed63672336b14886fede9d1*

6.4. Implementing getters of Card and Player

Before implementing the rest logic for game, we need also some more getters and setters for **Player** and **Cards**. So, we should implement them (this should be no problem for students since it was done before).

Commit: *034563a26c04c4eef2a0dea9cc57a3c309483df4*

6.5. Implementing supporting method for Player and fix in World

As the **Game** is now singleton, we have to add it to world in **MyWorld** class. The next task is to implement a supporting method for the player that will be used to receive some amount of damage. **Player** should decrease **Wall** or **Tower** according to it.

Commit: *e0d283ee339506ef34366a2b689aba1c17f6391c*

6.6. Implementing game logic

Finally, we have to implement game logic – **turn** method, that will consist of the following steps:

1. Check if one player wins – that means if either player tower reaches height 100 or falls under the value 0. If one of these conditions are met, we will display winning screen and exit game loop – call **return** statement.
2. Set active player to other one – just flip value of **isPlayer1Active** attribute.
3. Prepare cards for the next player – as we have already created method **prepareCards**, all we have to do in this step is to call it.
4. Handle players turn – specifically clicking on cards. As **Card** itself is able to listen to clicking on it, all we have to do is call **draw** method of game, which draws prepared cards.
5. Redraw players – this is done using method **redraw** for each player.

Then we need to implement the **useCard** method in **Game** using **switch**. This **switch** should contains branch for each card type and handle its execution. There are 7 card types: **BuildTower**, **BuildWall**, **IncreaseBricks**, **IncreaseSwords**, **Attack**, **IncreaseMagic** and **StealBricks**. Let us take a look at the first card type – **BuildTower**. In this case we have to check if player can play this card (i.e. number of his bricks is greater than or equal to card requirements), increase hit **Tower height** and decrease hit **bricksNumber** to card requirements. So the code can looks like this:

```
if (activePlayer.getBricksNumber() >= card.getRequirements())
{
      activePlayer.increaseTowerHeight(card.getEffect());
     activePlayer.setBricksNumber(
            activePlayer.getBricksNumber() - card.getRequirements()
      );
}
```

The other card types are similar:

- **BuildWall –** we have to check for players **bricksNumber** and we will increase **Wall height**.
- **IncreaseBricks** – we don't have to check for anything and will increase **bricksNumber**
- **IncreaseSwords** – we don't have to check for anything and will increase **swordsNumber**

- **`Attack`** – we have to check for players **`swordsNumber`** and call **`receiveDamage`** of inactive player
- **`IncreaseMagic`** – we don't have to check for anything and will increase **`magicNumber`**
- **`StealBricks`** – we have to check for **`magicNumber`**, decrease inactive player **`bricksNumber`** and increase active player **`bricksNumber`**.

It is possible to create also other card types – it is for students' imagination. These are some basic ones.

Finally, we have to call **`turn`** method after playing a card to ensure game logic.

As a last step, we should implement a winning screen. This is like other drawings in this project, so it is upon you to either create it with students or leave it for them.

To wrap it up, there are also some fixes in the **`Player`** and **`Tower`** for the sake of code cleaning.

Commit*: 9c91d4613fe0da1ff9c9960bf96fe7c27988fcf1*

4. Bibliography

[1] „Git,“ 1 10 2023. [Online]. Available: https://git-scm.com. [Cit. 1 10 2023].

[2] „GIT, SVN, mercurial – Google Trends,“ 2 10 2023. [Online]. Available: https://trends.google.com/trends/explore?cat=5&date=today%205-y&q=GIT,SVN,mercurial&hl=sk. [Cit. 2 10 2023].

[3] „GitHub: Let’s build from here · GitHub,“ 1 10 2023. [Online]. Available: https://github.com. [Cit. 1 10 2023].

[4] „The DevSecOps Platform | GitLab,“ 1 10 2023. [Online]. Available: https://about.gitlab.com. [Cit. 1 10 2023].

5. Attachments

5.1. Export of learning design for project Bomberman

See file LD_Bomberman.pdf

5.2. Export of learning design for project Tower defense

See file LD_Tower_defense.pdf

5.3. Export of learning design for project Ants

See file LD_Ants.pdf