NASTAVNI PLAN I PROGRAM POUČAVANJA PROGRAMIRANJA TEMELJEM OSNOVNIH NAČELA OBJEKTNO ORIJENTIRANOG PROGRAMIRANJA (“Lagani OOP”)

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

Projekt	Object Oriented Programming for Fun
Akronim projekta	OOP4FUN
Broj ugovora	2021-1-SK01-KA220-SCH-00027903
Projektni koordinator	Žilinska univerzita v Žiline (Slovakia)
Projektni partneri	Sveučilište u Zagrebu (Hrvatska) Srednja škola Ivanec (Hrvatska) Univerzita Pardubice (Češka Republika) Gymnazium Pardubice (Češka Republika) Obchodna akademia Povazska Bystrica (Slovačka) Hochschule fuer Technik und Wirtschaft Dresden (Njemačka) Gymnasium Dresden-Plauen (Njemačka) Univerzitet u Beogradu (Srbija) Gimnazija Ivanjica (Srbija)
Godina izdanja	2024

Sadržaj

Popis slika

Popis tablica

1. Uvod

1.1. Cilj predmeta

Cilj ovog predmeta je naučiti učenike kako rješavati problemske zadatke koristeći osnove objektno orijentiranog programiranja (OOP) prema temeljnim načelima OOP paradigme. Učenici će naučiti kako podijeliti zadatke među objektima koji surađuju, utvrditi njihove mogućnosti te implementirati dizajnirani model. Prethodno znanje programiranja nije nužno. Poučavanje se provodi korištenjem programskog jezika Java. U predmetu se objašnjavaju osnovni OOP koncepti (poput enkapsulacije, nasljeđivanja ili asocijacije) putem kreiranja računalnih igara gdje se spomenuti koncepti koriste na jednostavan i intuitivan način. Proces izrade računalne igre se temelji na timskom radu i praktičnoj primjeni znanja i vještina iz drugih područja informatike i njoj srodnih predmeta (rad s multimedijskim i uredskim programima). Dizajn svake računalne igre je dovoljno otvoren i slobodan tako da učenici mogu individualno i kreativno proširivati igru. Štoviše, dizajn dovodi do pravilne upotrebe stečenog znanja.

1.2. Karakteristike predmeta

Predmet je usmjeren na uvođenje inovativnog pristupa poučavanju programiranja, temeljenog na rješavanju zadataka korištenjem paradigme objektno orijentiranog programiranja (OOP). OOP je danas dominantna paradigma za razvoj aplikacija zbog čega je i primjereno da učenici posjeduju znanja i vještine iz ovog područja. U predmetu je predstavljeno razvojno okruženje koje koristi različite oblike uređivanja izvornog koda (uređivanje temeljeno na principu okvira – engl. *frame based programming*, ali i pisanjem izvornog koda) što omogućuje poučavanje učenika s različitim razinama prethodnog tehničkog znanja i aktivnosti. Svojom jednostavnošću i jasnoćom, ovaj alat podržava brzo i intuitivno razumijevanje nastavnih tema što pozitivno utječe na učenike i njihovu motivaciju.

1.3. Cilj predmeta

Programiranjem interaktivnih igara, učenik će steći određena znanja i vještine kojima će moći:

- identificirati problem,
- identificirati prikladne objekte za rješavanje identificiranog problema (dekompozicija objekta)
- dizajnirati klase objekata, njihove atribute i metode,
- identificirati i ispravno koristiti veze među objektima (asocijacija, nasljeđivanje),
- dizajnirati algoritam za rješavanje problema i distribuirati ga među objektima koji surađuju,
- koristiti elemente izvornog koda (grananje, petlje) za implementaciju dizajniranog algoritma,
- učinkovito koristiti sredstva za otklanjanje pogrešaka u izvornom kodu,
- izraditi jednostavnu aplikaciju s grafičkim sučeljem u Greenfoot okruženju.

1.4. Ishodi učenja

Ishodi učenja predmeta su sažeti na sljedeći način:

- razumijeti osnovne principe objektno orijentiranog programiranja,
- razumijeti osnove algoritmizacije,
- razumijeti sintaksu programskog jezika Java,
- analizirati izvršavanje program ana temelju izvornog koda,
- moći izraditi vlastiti program korištenjem OOP-a.

1.5. Materijalno-tehnički uvjeti

Računalna učionica treba sadržavati odvojeni radni prostor za svakog učenika i radni prostor za nastavnika. Radnim prostorom smatra se stol, stolica i osobno računalo (PC). Sva računala trebaju biti povezana na LAN mrežu s pristupom internetu (preporučeno).

Osobno računalo treba zadovoljavati sljedeće minimalne uvjete:

- operacijski sustav (Microsoft Windows 7 ili noviji, Linux (Debian), Mac OS 10.10 ili noviji),
- uredski softver s uređivačem teksta, tablica i prezentacija (npr. Microsoft Office, Libre Office, Open Office),
- Java SE Development Kit (JDK),
- Greenfoot razvojno okruženje (verzija 3.8 ili novija),
- jednostavan grafički softver,
- web preglednik (npr. Edge, Google Chrome, Mozilla Firefox, Opera),
- odgovarajući softver za ostale komponente računala.

2. Načela nastavnog plana i programa

Predloženi nastavni plan i program osmišljen je za rješavanje problema identificiranih u projektnim rezultatima (PR) PR1 i PR2 (vidi poglavlje "Usklađivanje rezultata s rezultatima PR1 u PR2 izvještaju). U sljedećoj tablici predstavljamo perspektivu razvoja kurikuluma, ishoda učenja, nastavnih materijala i nastavnih aktivnosti kako je predloženo u PR2. Slijedeći navedeno, uspjeli smo formulirati načela nastavnog plana i programa.

Tablica 1. Proizašla načela nastavnog plana i programa

PR2 rezultati	**PR3 načela nastavnog plana i programa**
U srednjim školama OOP treba uvesti na početku kroz teme koje pokrivaju osnovne pojmove programiranja, a uže teme vezane uz OOP bile bi prikladnije u zasebnim predmetima. Ključno je povezati i potaknuti razmjenu informacija između nastavnika na školskoj i sveučilišnoj razini, uz uključivanje kreatora politika koji definiraju nastavne planove i programe vezane uz vještine programiranja na svim obrazovnim razinama. Iz perspektive predmeta/predavača i iz perspektive dizajna predmeta, trebali bi se koristiti sljedeći inovativni oblici nastave/prijenosa znanja: kombinirano učenje, učenje kroz rad, rješavanje problema, problem suradnje, timski rad, učenje temeljeno na problemu, aktivno učenje, laboratorijsko učenje. Osim toga, u predavanjima, seminarima i laboratorijskim vježbama treba primjenjivati različite oblike inovativnih pristupa. Igre i igrifikacija općenito su se često koristili kao motivacija učenika za programiranje. Učenicima se svidjela prilika da budu kreativni ili da se natječu u znanju uz podršku odgovarajućeg okruženja. Iz rezultata pregleda literature predloženi su različiti načini učenja kroz igru: (1) učenje igranjem, (2) učenje stvaranjem igara, (3) učenje korištenjem alata povezanih s igrama i (4) učenje pomoću igrifikacije.	Nastavni plan i program mora pravilno koristiti OOP od samog početka slijedeći “object-first” pristup. Prikladna razina OOP-a za srednje škole identificirana je i formulirana kao lagani OOP. Lagani OOP može imati prednost pri stvaranju igara, s obzirom na to da je lako identificirati objekte u igri, kao i mogućnosti i svojstva objekata. Kako bi se zainteresiralo učenike za korištenje igara, potrebno je zainteresirati učenike za igre općenito. Trebalo bi izraditi nekoliko projekata temeljenih na igrama s različitim principima bi se ispunio ovaj cilj. Štoviše, nekoliko igara koje će se pripremiti omogućit će: • kreiranje različitih nastavnih materijala usmjerenih na različite oblike nastave (online/onsite, intenzivni/cjelogodišnji oblik, početni/napredni učenici). Ovo je ključno svojstvo za nadolazeće projektne rezultate, • kreiranje jedne igre uz prisustvo nastavnika, a ostale igre kroz domaće uradke (nastavnik će moći vidjeti mogu li učenici primijeniti znanje u drugačijem kontekstu), • kreiranje igre prema danim uputama uz minimalnu interakciju nastavnika, tako da učenici mogu razumjeti važnost dobro napisanog tehničkog opisa i primijeniti naučene vještine za kreiranje igre kako je navedeno, • uvođenje novog koncepta laganog OOP-a kako igra uvodi nove pojmove. To će omogućiti zaustavljanje razvoja projekta ako se učitelj odluči pokriti podskup laganog OOP-a zbog specifičnosti nacije.
Za učenje i poučavanje OOP koncepata, učenje stvaranjem igara pokazalo je značajne učinke u poboljšanju vještina učenika u rješavanju problema i njihovo uključivanje u zabavno okruženje.	

PR2 rezultati	PR3 načela nastavnog plana i programa
Kao što je istaknuto u nekoliko PR2 rezultata, glavni cilj bi trebao biti uključivanje zadataka učenja i poučavanja u poticajne i zabavne aktivnosti koje će se pozitivno odraziti na povećanje interesa pohađanja i završenosti. To bi trebalo općenito povećati interes srednjoškolaca za programiranje i na kraju dovesti do boljeg razumijevanja programiranja i OOP koncepata. U tom slučaju učenici ne bi bili "izgubljeni" kada bi se suočili sa sveučilišnim kurikulumom.	Projekti će se graditi na principu učenja kroz rad. Pokušat ćemo minimizirati teoretsko objašnjavanje i podržati istraživačke, praktične i proizvodne aspekte učenja. Rad u grupama ohrabruje učenike koji bi u početku mogli imati problema. Ako se projekt razvija u grupi, mogu se koristiti agilne metode razvoja, ali i nastave. Prije provedbe, nastavnik može odlučiti koristiti analitičku i dizajnersku fazu projekta. To će omogućiti korištenje prethodno stečenih znanja i vještina učenika kao i uključivanje učenika u izradu projekata. Od učenika se može tražiti da: • rade s izvorima informacija kako bi pronašli pravu igru, • formuliraju pravila igre i/ili zahtjeve za njihovu primjenu (u usmenom ili pisanom obliku), • pripreme multimedijalne sadržaje (slike/zvukove).
Glavni cilj PR2 je bio pronaći prikladne i inovativne ideje učenja i poučavanja te pristupe koji bi riješili navedene probleme. Kao što je spomenuto u prethodnim poglavljima[1], postoji nekoliko identificiranih dobrih praksi koje bi se mogle koristiti za poboljšanje postignuća ishoda učenja tijekom srednjoškolskog obrazovanja. Međutim, treba napomenuti da bi redizajn kurikuluma trebao rezultirati uvođenjem OOP tema i postavljanjem ciljeva za postizanje ishoda učenja povezanih s OOP-om. Također je važno odabrati i odgovarajuću vrstu vrednovanja: (online) upitnici jedina su prihvaćena metoda za procjenu učeničkog zadovoljstva, korisnosti, interesa, angažmana i pojednostavljenja programskih i OOP koncepata koji će se definirati na srednjoškolskoj, a ne na sveučilišnoj razini.	S naglaskom na princip učenja kroz rad, pokušali smo minimizirati akvizicijski tip aktivnosti i ojačati istraživački dio. Projekti će se izgraditi na način koji će omogućiti jednostavno proširivanje kako bi motivirani učenici mogli samostalno raditi na projektima. Kako bismo potvrdili predloženi kurikulum, pripremit ćemo dizajn učenja za svaki projekt. Nakon toga ćemo usporediti analitički rezultat novih projekata koji koriste lagani OOP s analitičkim rezultatom dizajna učenja izgrađenog za projekt koji je razvijen u prošlosti i već se primjenjuje u praksi u Slovačkoj (između ostalih škola u zemlji, koristi ga projektni partner Obchodna akademia Povazska Bystrica) i Češkoj (koju koristi projektni partner Gymnazium Pardubice). Pozitivne povratne informacije za ovaj potvrđeni projekt objavljene su u PR2 te pretpostavljamo da ćemo, slijedeći ista najbolja načela i prakse, prenijeti pozitivno prihvaćanje predloženog kurikuluma.
Primjenom timskog rada u OOP zadacima, učenici bi imali priliku podijeliti svoje znanje i prenijeti implementaciju osnovnih koncepata	Prilikom dizajniranja ovdje predloženih projekata temeljenih na igrama, imali smo na umu korištenje različitih metoda, kao što je

[1] pogledati prethodna poglavlja PR2 izvještaja

<table>
<tr><th>PR2 rezultati</th><th>PR3 načela nastavnog plana i programa</th></tr>
<tr><td>programiranja na druge učenike - peer-to-peer (ravnopravno) učenje.

Rezultati ovog projekta urodit će skupom materijala koji će dati priliku visoko motiviranim učenicima sa solidnim predznanjem da to znanje prošire kroz različite aktivnosti i uloge. Takvi bi učenici mogli značajno poboljšati ukupna postignuća cijele grupe ako im se pruži prilika da podijele znanje ili da vode timove.</td><td>EduScrum. Zbog toga svaki projekt isporučujemo u obliku GIT repozitorija i pružamo odgovarajuće znanje nastavnicima. Iako korištenje ovog pristupa nije neophodno pa učitelji i dalje mogu koristiti tradicionalni pristup, korištenje agilnih metoda dovest će do poučavanja:
• osnova sustava za izradu verzija - predlažemo GIT kao najkorišteniji sustav za izradu verzija, a korištenje sustava za izradu verzija olakšava
o dijeljenje izvornih kodova i
o razvoj novih značajki igara bez utjecaja na tijek projekta (npr. u posebnoj grani), što će motivirati ambiciozne učenike.
• timski rad – svaki učenik će biti odgovoran za određeni aspekt igre što dovodi do potrebe za učinkovitom komunikacijom između članova tima.
• upravljanje vremenom – pojedini dijelovi projekata morat će se isporučiti na vrijeme kako bi se spojili i kako bi se nastavilo s radom, međutim, pravilna uporaba sustava za izradu verzija i OOP-a otvara mnogo mogućnosti za rješavanje problema s vremenom što može dovesti do pozitivne motivacije učenika čak i tijekom zahtjevnih razdoblja.
S nastavnicima ćemo realizirati događaje s multiplicirajućim učinkom koji će pokrivati kako uvod u GIT tako i korištenje principa agilnog razvoja softvera u nastavi. Učenici će formirati projektne timove s određenim ulogama, dijelit će ideje na *stand-up*-u, zadavati si ciljeve, realizirati sprinteve, izrađivati dokumentaciju i druge artefakte te prezentirati svoje rješenje.</td></tr>
<tr><td>Potrebno je poticati korištenje više različitih alata i programskih jezika u ranijim godinama učenja. Koncepti programiranja mogu se pojednostaviti korištenjem alata za vizualizaciju. Iako su neke zemlje već uvele korištenje alata poput Logo-a ili Scratch-a, oni su interesantni osnovnim, ali ne i srednjim školama. Stoga bi se trebali koristiti napredniji alati koji su dizajnirani za podršku OOP-u. Prepoznali smo da se Alice i Greenfoot ističu među ostalim alatima.</td><td>Koristimo Greenfoot okruženje koje primjenjuje Java programski jezik. Java je trenutno vrlo popularan i u praksi široko korišten programski jezik.

Štoviše, Greenfoot predstavlja frame-based uređivač izvornog koda koristeći Stride jezik. Ovo otvara mogućnosti za nastavnike koji će metode predstavljene u ovom nastavnom planu i program htjeti koristiti s učenicima mlađe dobi.</td></tr>
</table>

PR2 rezultati	PR3 načela nastavnog plana i programa
	Greenfoot je vrlo vizualan i od samog početka omogućuje stvaranje vizualiziranog objekta koji je "živ" i s kojim se može komunicirati. Stoga je teorijski uvod sveden na minimum, a učenici će početi raditi od samog početka.
Kako su učenici naveli, za poučavanje programiranja važno je da nastavnici koriste nove i suvremene nastavne materijale i koriste kreativne metode poučavanja. Također, dostupnost i fleksibilnost nastavnika za rad s učenicima izvan učionice neophodna je za motivaciju učenika i stvaranje većeg interesa za predmet.	Koristeći predstavljena načela, izrađujemo moderan nastavni plan i program za nastavnike koji će pokrivati lagane OOP teme i temelji se na projektnom radu te koristi object-first pristup. Predlažemo nekoliko projekata temeljenih na igrama od kojih je svaki isporučen u obliku GIT repozitorija. Za svaki projekt izrađen je dizajn učenja koji je omogućio validaciju ovih projekata s već korištenim i pozitivno prihvaćenim pristupom u praksi.

Sadržaj i opseg obrazovnog programa razlikuju se s obzirom na projekt igre koji se razvija tijekom nastave. Za detaljnu analizu pogledajte odgovarajući dizajn učenja i priložene datoteke s analizom.

3. Projekti

Što se tiče principa nastavnog plana, stvorena su dva projekta igara koji pravilno koriste principe objektnog programiranja i učenja kroz praksu. Osim toga, obrađen je projekt Bomberman, koji je bio osnovni projekt nacionalnog projekta IT Akademije, koji se koristi u praksi u Slovačkoj. Koristimo ovaj projekt za validaciju predloženih projekata. Organizacija svih poglavlja projekta je kako slijedi.

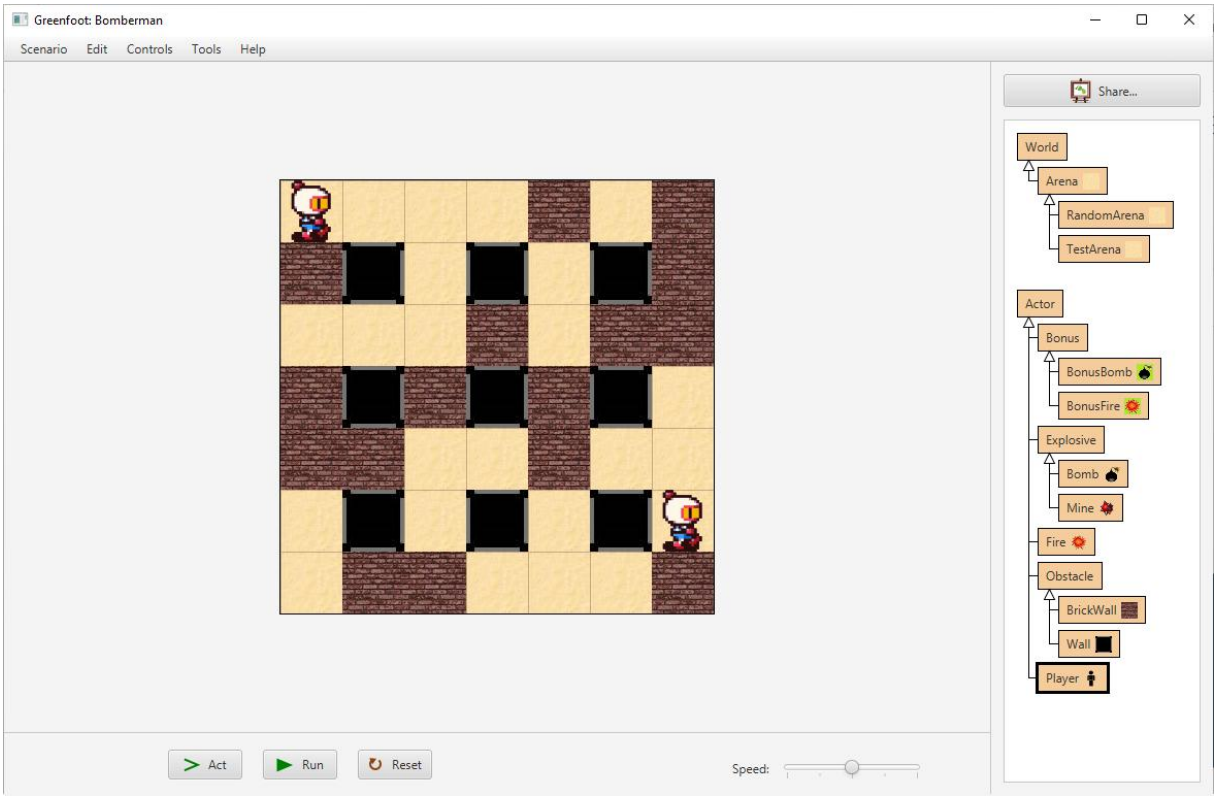
- **Opis projekta -** osnovni opis igre s prikazom završene aplikacije i sažetim pravilima igre. Opis projekta također predstavlja povezanost s osnovnim OOP temama.
- **Link na izvorne kôdove** - izvorni kôdovi su organizirani u obliku GIT [1] repozitorija. Korištenje pravilno upravljanih repozitorija omogućuje nastavniku korištenje suvremenog pristupa upravljanju izvornim kôdom. Koristimo GIT kao verzionirajući sustav koji je među najpopularnijima u posljednjih nekoliko godina [2]. GIT nam također omogućuje korištenje repozitorija u oblaku, poput GitHub [3] ili GitLab [4], koji su besplatni i nude korištenje mnogih alata za poboljšanje suradnje timova. Svaki repozitorij je organiziran kako slijedi:
    - **Grana po temi -** zadaci svake teme iz nastavnog plana razvijaju se u posvećenoj grani. Glavna grana sadrži samo inicijalnu verziju koda i spajanja grana teme.
    - **Verzija koda po zadatku -** svaki zadatak koji je usmjeren na stvaranje/modifikaciju izvornog kôda je u obliku verzije koda (engl. commit). Opis verzije odgovara broju odgovarajućeg zadatka
- **Link na dizajn učenja -** nastavni plan izrađen je u obliku dizajna učenja. Dizajn učenja omogućuje nam definiranje ishoda učenja (koji pokrivaju potrebne kompetencije identificirane u PR1 i PR2) i povezivanje s temama (vidi povezanost s granama odgovarajućeg GIT repozitorija). Teme su organizirane u jedinice koje su sastavljene od TLAs (vidi povezanost s verzijama koda u odgovarajućem GIT repozitoriju). Korištenje dizajna učenja omogućuje analizu raspodjele vremena što je izravno povezano s validacijom deklariranog principa učenja kroz praksu. Budući da je dizajn učenja obrađen i za već uspostavljeni projekt koji se koristi u pedagoškoj praksi (Bomberman), omogućuje nam identifikaciju potencijalnih problema u dizajnu. Kako bismo mogli provesti validaciju, projekti su organizirani po temama na isti način.
- **Obuhvaćene teme osnovnog OOP-a**
- **Sadržaj i opseg obrazovnog programa** - pregled opterećenja učenika u određenom tipu učenja kao i pregled doprinosa tema pojedinim ishodima učenja.
- **Popis tema**. Svaka tema sadrži:
    - kratak opis,
    - usporedbu dizajna učenja,
    - popis zadataka.

3.1. Bomberman

Bomberman je prilično poznata višeigračka (engl. *multiplayer*) igra. Igra se odvija na areni koja sadrži igrače kao i neke fiksne prepreke. Igrač može postaviti bombe koje eksplodiraju nakon određenog

vremena. Cilj igre je eliminirati protivnike koristeći bombe. Neke od prepreka na areni mogu se uništiti korištenjem bombi. Nakon uništenja prepreke, u igri može se pojaviti slučajni bonus, koji na primjer povećava brzinu ili snagu bombi igrača. Igra završava ako u igri ostane samo jedan igrač, u kojem slučaju taj igrač pobjeđuje, ili ako u igri više nema igrača, u kojem slučaju igra završava neriješeno.

Projekt Bomberman obuhvaća većinu tema osnovnog OOP. Fokusira se na glavne aspekte OOP-a koji su sažeti u temama u nastavku. Nadalje, uvodi neke teme koje idu izvan lagane OOP, poput generiranja i korištenja slučajnih vrijednosti korištenjem klase **Random**.



Slika 1: Okruženje Greenfoot s konačnim stanjem projekta Bomberman

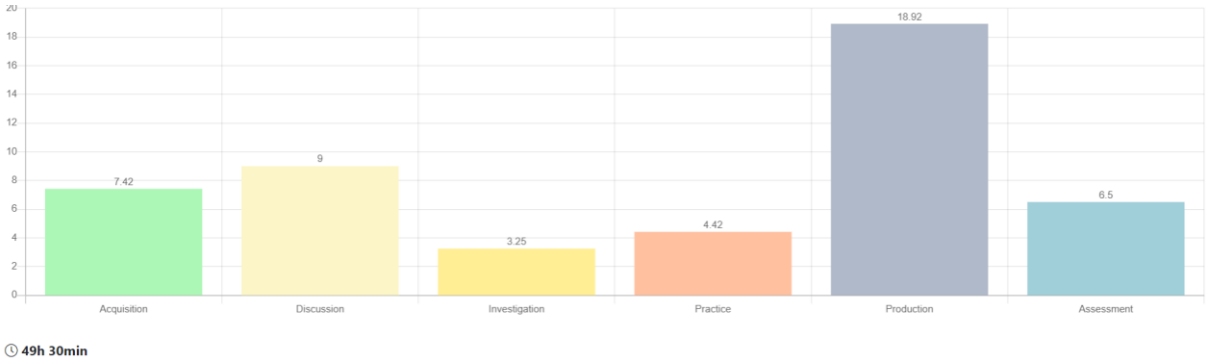
Izvorni kôd je dostupan na:
https://gitlab.kicon.fri.uniza.sk/oop4fun/project-bomberman

Dizajn učenja dostupan je na:

https://learning-design.eu/en/preview/70bcf65d805b0603f6c1aeab/details

3.1.1. Sadržaj i opseg obrazovnog programa

Ukupno opterećenje učenika iznosi 49 sati i 30 minuta i raspoređeno je kako slijedi:



Slika 2: Opterećenje učenika prilikom korištenja projekta Bomberman

Konstruktivno usklađivanje je sažeto u tablici u nastavku:

Tablica 2. Konstruktivno usklađivanje projekta 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						

Za detaljan plan pogledajte prilog 5.1.

3.1.2. Teme

Projekt Bomberman je podijeljen na deset tema:

Obuhvaćene teme osnovnog OOP-a su:

- klase, objekti, instance
- metode, proslijeđivanje argumenata metoda
- konstruktori
- atributi
- enkapsulacija
- nasljeđivanje
- apstraktne klase
- životni ciklus objekta

1. Uvod u okruženje Greenfoot-a

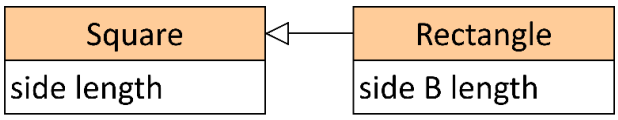
Tema je posvećena osnovnom postavljanju projekta. Učenici će naučiti kako postaviti dimenzije i izgled okruženja, stvoriti klasu (kao podklasu klase **Actor**), stvoriti njezinu instancu, poslati joj poruku i promatrati njezino unutarnje stanje.

1.1. Identificiraj objekte

Identificiraj objekte u svojoj okolini i navedi njihove svojstva i radnje koje mogu izvršavati. Možete li identificirati objekte koji nemaju svojstva? Možete li identificirati objekte koji ništa ne mogu raditi? Možete li identificirati nematerijalne objekte (one koje ne možemo fizički dodirnuti)?

1.2. Validiraj hijerarhiju objekata

Sljedeća slika prikazuje hijerarhiju klasa Square (Kvadrat) i Rectangle (Pravokutnik). Je li ovo dobra hijerarhija?



1.3. Kreiraj jednostavnu hijerarhiju

Kreiraj hijerarhiju klasa:

a) prijevoznih sredstava
b) životinja,
c) i dijelova računala.

Navedi svojstva koje svaka klasa definira. Navedi svojstva koja definira svaka klasa. Koje klase, po svom dizajnu, predstavljaju klase koje se ne može materijalizirati, kao da ih ne možemo dotaknuti? Takve klase ćemo zvati *apstraktnim klasama.*

1.4. Kreiraj pločicu

U grafičkom uređivaču, kreiraj pločicu koja će grafički predstavljati ćeliju svijeta. Odaberi kvadratnu reprezentaciju, idealno 60x60 piksela. Uvezi ili spremi sliku u mapu projekta u mapi slika. Postavi sliku kao sliku svijeta. Imajte na umu da je **MyWorld** klasi u dijagramu razreda dodana mala ikona koja predstavlja njezinu grafičku formu.

1.5. Kreiraj svijet

Izmijeni konstruktor klase **MyWorld** kako bi stvorio svijet od 25x15 ćelija, pri čemu je svaka ćelija veličine 60 piksela. Kako bi slika trebala biti izmijenjena kako bi svijet izgledao kao šahovska ploča (tj. s izmjenično obojenim kvadratima)?

Zatim, kreiraj klasu **Player**. To se radi desnim klikom na klasu **Actor** i odabirom opcije **New subclass...**. Nazovi podklasu **Player** i odaberi njezinu sliku iz biblioteke. Opet, primijetite malu ikonu pored klase **Player**, slično kao kod klase **MyWorld**. Da biste stvorili instancu klase **Player**, desnim klikom na klasu **Player** odaberite **Novi Player**, a zatim premjestite ikonu s igračem na pozadinu svijeta. Lijevim klikom postavite instancu. Da biste pregledali status instance, desnim klikom na nju odaberite **Inspect**.

1.6. Pregledaj stanje igrača

Uhvatite stvorenu instancu klase **Player** mišem i premjesti je na drugu poziciju u svijetu. Promatrajte stanje unutar - što vidite? Kreirajte još jednu instancu klase **Player** i također pregledajte njezino unutarnje stanje. Ponovno povucite jednu od dvije instance mišem - koje se unutarnje stanje promijenilo?

1.7. Interakcija s igračima

Pozovite metode koje pruža Greenfoot okruženje nad različitim instancama klase Player. Da biste ovo napravili, desnom tipkom miša kliknite na instancu i odaberite, na primjer, metodu **void move(int)**. Po upitu, upišite cijeli broj. Promatrajte kako se unutarnje stanje instance mijenja.

2. Algoritam, upravljanje aplikacijom, stvaranje metoda

Ova tema obuhvaća stvaranje javnih metoda koji pomiču igrača u svijetu. Također uvodi alate za upravljanje izvođenjem scenarija u Greenfoot okruženju.

2.1. Napiši jednostavan algoritam

Zapišite postupak, kako pripremiti kavu, kako putovati do škole i kako skuhati ručak.

2.2. Napiši općenitiji algoritam

Napravite općeniti algoritam za pripremu toplog napitka. Razmislite o tome što su ulazi takvog algoritma kako bi bio općenit.

2.3. Pregledaj instancu klase

Istražite metode instance klase **Player**. Da bi ovo napravili, desnom tipkom miša kliknite na klasu i odaberite narebdu **Open Editor**. Što primjećujete? Po analogiji prema metodi **act()** dodajte **makeLongStep()** metodu.

2.4. Implementiraj metodu

Dodajte izjavu u tijelo metode **makeLongStep()** tako da se instanca klase Player pomakne za dvije ćelije u trenutnom smjeru. Zatim stvorite više instanci klase Player i pozovite ovu metodu na svakoj instanci. Je li ponašanje očekivano?

2.5. Dodaj dokumentaciju

Dodajte dokumentacijski komentar za metodu **makeLongStep()**.

2.6. Dodaj više dokumentacije

Uredite dokumentacijski komentar klase **Player**. Dodajte verziju klase i njenog autora.

2.7. Pročitaj dokumentaciju

Istražite prozor dokumentacije.

2.8. Dodajte akciju igrača

Izmijenite tijelo metode **act()** klase **Player** da bi pozvalo metodu **makeLongStep()**.

2.9. Istraži kontrole aplikacije

Isprobajte gumbe koji kontroliraju aplikaciju. Stvorite više instanci klase **Player**. Pritisnite gumb **Act** - što se događa? Pritisnite gumb **Run** - što se događa? Nakon pritiska na gumb **Run** pritisnite gumb **Pause**, što se događa? Koje djelovanje ima klizač **Speed** na poziv metode **act()** nakon što je pritisnut gumb **Run**? Što se dogodi kada klikneš na gumb **Reset**?

2.10. Dodaj drugu akciju igrača

Dodajte metodu klasi **Player** koja će šetati instancom klase **Player** u kvadratu. Dokumentirajte svoju metodu. Koristite odgovarajuće metode iz bazne klase **Actor** za kretanje i rotaciju. Izmijenite metodu **act()** tako da instanca klase **Actor** hoda u kvadratu kada je pozvana. Zatim provjerite svoje rješenje pokretanjem aplikacije.

3. Grananje i kontrola igrača

Ova tema uvodi učenike u grananje u obliku **if-else** izjave i **switch** izjave. Grananje se koristi u otkrivanju rubova svijeta i sudara s zidovima - koji su novi objekti dodani u ovu temu.

3.1. Pomakni igrača

Izmijenite kôd metode **act()** u klasi **Player** tako da se igrač kreće samo kada je pritisnuta tipka **M** (M kao kretanje). Zadržite kôd odgovoran za okretanje igrača kada dosegne rub svijeta, ali razmislite o njegovom položaju. Kada se može izvesti okretanje igrača?

3.2. Promatraj stanje igrača

Stvorite instancu klase **Player** i smjestite je u središte ploče. Otvorite prozor s unutarnjim stanjem instance i pozicionirajte ga tako da je vidljiv dok aplikacija radi. Zatim pokrenite aplikaciju i promatrajte kako se mijenjaju vrijednosti atributa **x** i **y** u klasi **Player**. Kako se ove vrijednosti mijenjaju dok se krećete gore, dolje, lijevo i desno? Koristi različite vrijednosti za **setRotation()** metodu (0, 90, 180, 270) i pokušaj zamijeniti **isAtEdge()** metodu sa **getX()** i **getY()** metodama.

3.3. Dodaj detekciju ruba svijeta

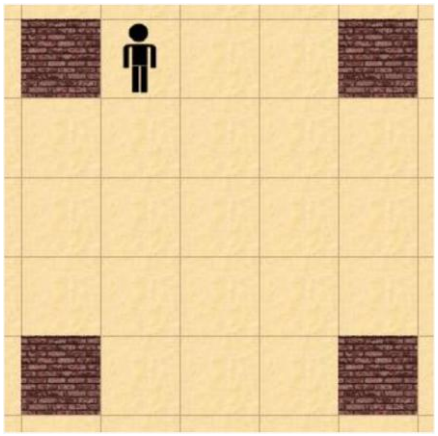
Dodajte kôd u tijelo metode **act()** da se igrač pravilno okrene nakon što dosegne dno i lijeve rubove svijeta.

3.4. Dodaj zidove

Stvorite dvije nove klase kao podklase klase **Actor**. Prva klasa bit će **BrickWall** klasa, a druga klasa bit će **Wall**. Pripremite prikladne slike veličine 60x60 piksela u grafičkom uređivaču. Zatim dodijelite te slike novostvorenim klasama.

3.5. Promatraj kretanje igrača

Stvorite četiri instance klase **BrickWall** i jednu instancu klase **Player** kako je prikazano na slici. Pogodite kako će se igrač kretati? Pokrenite aplikaciju. Odgovara li vaše predviđanje onome što promatrate?



3.6. Dodaj detekciju sudara sa zidom

Dodajte kôd u metodu **act()** klase **Player** kako biste osigurali da se igrač okrene za 90° suprotno od kazaljke na satu kada uđe u ćeliju koja sadrži instancu klase **Wall**.

3.7. Predvidi kretanje igrača

Predvidite kako će se instanca klase **Player** kretati. Odgovara li rezultat vašoj predikciji?

3.8. Promatraj otkrivanje ruba

Postavite jednu instancu klase **Player** u kutove svijeta. Predvidite kako će se ova instanca kretati kada se pokrene aplikacija. Odgovara li rezultat vašoj predikciji?

3.9. Dovrši detekciju sudara sa zidom

Dopunite kaskadu uvjeta kako biste provjerili dodir s instancom klase **BrickWall** i klase **Wall** samo ako se igrač ne nalazi na rubovima svijeta. Prvo provjerite instancu klase **BrickWall**.

3.10. Pomiči igrača automatski

Stvorite metodu za automatsko pomicanje instance klase **Player**. Premjestite sav kôd iz metode **act()** u novu metodu. Naziv metode može biti, na primjer, **moveAutomatically**.

3.11. Pomiči igrača korištenjem strelica

Stvorite metodu **moveUsingArrows()** u klasi **Player**. Programirajte ovu metodu tako da se igrač kreće samo kada je pritisnuta strelica. Kretat će se u smjeru pritisnute strelice. Pazite da kôd bude učinkovit. U metodi **act()** pozovite novu metodu.

3.12. Priprema slika

Pripremite četiri slike za igrača da se kreće gore, dolje, lijevo i desno. Dimenzije slika ne smiju prelaziti dimenzije ćelije, u našem slučaju 60x60 piksela. Smjestite pripremljene slike u direktorij **images**.

3.13. Upotreba slika

Kreirajte metodu **updateImage()** koja mijenja sliku igrača prema njegovoj trenutnoj rotaciji. Dodajte poziv ovoj metodi u tijelo metode **act().**

3.14. Pokretanje aplikacije

Pokrenite aplikaciju. Primijetite da se slike prilagođavaju, ali se okreću prema tome kako je igrač okrenut. Riješite ovaj problem.

3.15. Upotreba više igrača

Pokušajte stvoriti više igrača i upravljati njima pomoću tipkovnice. Što primjećujete?

4. Varijable, izrazi i napredna kontrola igrača

Ova tema se bavi uvođenjem varijabli u obliku atributa klase i parametara metode. će koristiti ove atribute za kontrolu brzine igrača, kao i postavljanje specifičnih tipki koje kontroliraju kretanje igrača. To će omogućiti igri da ima više igrača, pri čemu će svaki biti kontroliran različitim tipkama.

4.1. Primijetite razliku

Koja je razlika između sljedećih algoritama?

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. Napiši jednostavne izraze

Napišite izraz koristeći logičke i relacijske operatore koji će izraziti da varijabla tipa int a ima vrijednost koja pripada sljedećim intervalima: **<-10,10), (5,142), (-11,-3) OR (1,25>**.

4.3. Evaluiraj izraze

Odaberite vrijednosti varijabli **x** i **y** i dodajte ih izrazima. Koje će vrijednosti imati varijable **b1** i **b2** u svakom slučaju (1 i 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);
```

U klasu **Player** dodaje **String** atribute **upKey**, **downKey**, **rightKey** i **leftKey**. To će biti tipke koje će se koristiti za kontrolu kretanja instance klase **Player**. Ključna riječ **private** znači da će atributi biti dostupni samo unutar klase **Player**. Unutar ove klase, atributi će biti pristupani preko riječi **this** (tj. instance ove klase), tj. **this.upKey**, **this.downKey**, **this.rightKey** i **this.leftKey**. Nadalje, modificira se metoda **moveUsingArrows()** tako da se tipka **"left"** zamijeni s atributom **this.leftKey**. Ostale tipke su na sličan način zamijenjene. Tipke za kontrolu kretanja dane instance moraju biti specificirane prilikom stvaranja instance. Za to je potrebno kreirati konstruktor. Konstruktor je posebna metoda za stvaranje instance određene klase, koja se izvršava prilikom stvaranja instance. Konstruktor opet ima četiri parametra - **upKey**, **downKey**, **rightKey** i **leftKey**. Unutar konstruktora treba razlikovati **leftKey** i **this.leftKey**. Prvi je parametar konstruktora, dok je drugi atribut instance klase. Konstruktor nije obavezan unutar klase. Ako ga ne implementiramo, koristit će se zadani konstruktor s praznim kodom.

4.4. Testiraj konstruktor

Testirajte konstruktor i modificiranu metodu za kontrolu kretanja igrača umetanjem dvije instance klase **Player** u svijet. Za svaku instancu u dijalogu postavite različite tipke za kontrolu njezinog kretanja. Testirajte možete li kontrolirati umetnute igrače neovisno jedan o drugome.

4.5. Preimenuj klasu

Preimenujte klasu **MyWorld** u **Arena**. Imajte na umu da ime konstruktora mora biti isto kao ime klase. Testirajte kako Greenfoot okruženje reagira ako nije tako. Dosad je uvijek bilo potrebno ručno dodavati instance klase, koje su nestajale nakon klika na gumb **Reset**. To je zato što u konstruktoru klase **Arena** nisu bile kreirane nikakve instance.

4.6. Dodaj još jednoj igrača

Dodajte još jednog igrača u svijet, na primjer koristeći atribut reference **player2**, koji će biti kontroliran tipkama **w** - gore, **s** - dolje, **d** - desno i **a** - lijevo. Postavite igrača na koordinate **[24,14]**. Nakon dodavanja, testirajte kontrole.

4.7. Reference atributa

Što bi se dogodilo ako bismo napravili dodjelu **this.player1 = this.player2**; nakon što su stvoreni oba igrača? Bi li to bio problem?

4.8. Proširi klasu igrača

Proširite klasu **Player** dodatnim atributom tipa **int** koji predstavlja veličinu koraka igrača. Ovdje se koriste dva konstruktora, koji se razlikuju u broju parametara. To je tzv. preopterećeni konstruktor. Koristi se onaj s odgovarajućim parametrima.

4.9. Integriraj veličinu koraka

Izmijenite metodu **moveUsingKeys()** kako biste poštovali novi atribut za veličinu koraka. Stvorite novu instancu klase **Player** i testirajte funkcionalnost programa.

4.10. Postavi igrače da se kreću različitim brzinama

Vaš zadatak je osigurati da se igrač kreće po jednu ćeliju, ali različitim brzinama. Svaki igrač može imati različitu brzinu. Kao podsjetnik, možete programirati različite brzine kretanja, na primjer, ne krećući igrača svaki put kad se detektira pritisak tipke (detekciju vršimo u metodi **act( )**, pa ako držite pritisnutu tipku, detektirat ćete njezin pritisak svaki put kada se izvrši), već samo svaki N-ti put kada se izvrši. Što veći N, to će niža biti brzina igrača. Kako ćete unijeti N? Gdje ćete ga pohraniti? Kako ćete znati koliko je prošlo pritisaka tipke? (Pomoć: potreban je i brojač).

5. Suradnja objekata i klasa

Glavni fokus ove teme je suradnja objekata. U ovoj temi, učenici će dodati interakciju između objekata u areni u projekt - na primjer, osiguravajući da igrač ne može proći kroz zidove. Nadalje, u ovoj temi, učenici će također dodati objekt bombe - jedan od glavnih dijelova igre Bomberman - u projekt.

5.1. Dodaj kôd za vertikalno kretanje

Analogno, dodajte kôd za kretanje gore i dolje na isti način (tako da ćete promijeniti vrijednost lokalne varijable **y**).

5.2. Provjeri mogućnost kretanja

Izmijenite metodu koja osigurava kretanje igrača (**moveUsingArrows**) kako biste provjerili mogućnost ulaska u ciljnu ćeliju prije promjene položaja igrača.

5.3. Razmotri ciglene zidove

Izmijenite metodu **canEnter()** tako da igrač također reagira na instance klase **BrickWall** i ne može proći kroz njih.

5.4. Dodajte bombu

Stvorite novu klasu **Bomb** i dizajnirajte njene atribute tako da predstavljaju snagu eksplozije. Stvorite parametarski konstruktor i inicijalizirajte atribute objekta.

5.5. Provjeri mogućnost postavljanja bombe

Dodajte metodu **canPlantBomb()** u klasu **Player** s povratnom vrijednošću tipa **boolean**, koja vraća zastavicu koja govori je li moguće postaviti bombu na ćeliju na kojoj se igrač trenutno nalazi. Bomba se može postaviti kada se pritisne odgovarajuća tipka i nema druge bombe na ćeliji.

5.6. Dodaj zvučne efekte

Proširite igru tako da eksplozija bombe bude popraćena zvučnim efektom. Zvuk može biti snimljen ili preuzet s interneta. Pronađite naredbu za reprodukciju zvuka u dokumentaciji Greenfoot klase.

6. Naslijeđivanje i for petlja

Ova tema bavi se uvođenjem naslijeđivanja. Učenici stvaraju nadklasu za klase **Wall** i **BrickWall**. Zatim će stvoriti testnu arenu kao podklasu klase **Arena**. Konačno, ovaj dio fokusira se na petlje s fiksnim brojem ponavljanja.

6.1. Dodaj nadređenu klasu

Kreirajte klasu **Obstacle**. Koja je nadklasa klase **Obstacle**? Uredite zaglavlja klasa **BrickWall** i **Wall** tako da budu podklase klase **Obstacle**.

6.2. Pojednostavi testiranje zauzetosti

Nakon dodavanja nadređene klase **Obstacle**, lakše je testirati može li igrač ući u određenu ćeliju. U svojoj metodi **getObjectsAt()**, svijet zahtijeva kao treći parametar klasu koju treba potražiti na određenoj ćeliji. Budući da su **i BrickWall** i **Wall Obstacle**, mogu se tretirati jednako. Izmenite metodu **canEnter()** u klasi **Player** da koristi samo jedan popis prepreka.

6.3. Modificiraj klasu Arena

Modificirajte klasu **Arena** tako da njen konstruktor ima dva parametra koji predstavljaju širinu i visinu. Modificirajte poziv konstruktora nadklase u klasi **Arena** kako bi uzeli ove parametre. Primijetite da se Arena ne može automatski konstruirati pomoću Greenfoota, budući da su joj potrebni parametri za konstruktor. Uklonite iz ovog konstruktora kôd koji je odgovoran za stvaranje i postavljanje igrača u arenu, to će biti učinjeno od strane podklasa. Također možete ukloniti deklaraciju atributa tipa **Player** iz klase **Arena**.

6.4. Popravi klasu TestArena

Modificirajte konstruktor klase **TestArena** tako da stvori praznu arenu veličine 7x7 ćelija. Kako biste provjerili, kreirajte instancu klase **TestArena** - iz kontekstnog izbornika klase **TestArena** odaberite stavku **new TestArena()**.

6.5. Dodaj upit za dimenzije

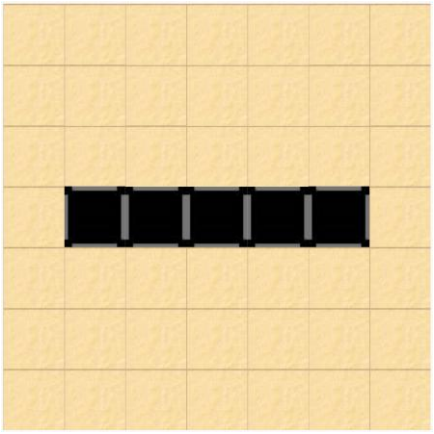
Dodajte metodu **showDimensions()** klasi **TestArena** koja će ispisati dimenzije arene na ekranu.

6.6. Dodaj još jedan upit za dimenzije

Dodajte metodu **showDimensions()** klasi **Player** koja će prikazati dimenzije arene ako je u testnoj areni – klasa **TestArena**. Koristite metodu **showDimensions()** klase **TestArena.**

6.7. Dodaj zidove u testnu arenu

Modificirajte konstruktor klase **TestArena** kako bi stvorili arenu veličine 7x7 ćelija s postavljenim zidovima kako slijedi:



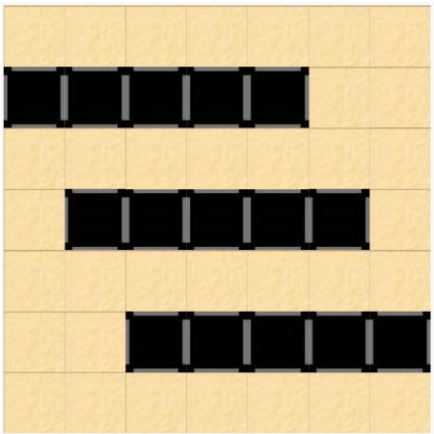
Posjetimo se da možemo koristiti metodu **addObject()** klase **World**, koja ima tri parametra, kako bismo umetnuli instancu klase **Actor** u svijet (tj. u podklase klase World i, u našem slučaju, klasu **Arena**):

- actor koji se treba umetnuti,
- x-koordinatu ćelije koja se umetne (tj. indeks stupca koji se numerira od 0),
- y-koordinatu ćelije koja se umetne (tj. indeks retka koji se numerira od 0)

Pozicija [0;0] u svijetu je u gornjem lijevom kutu. Budući da želimo odjednom dodati pet instanci klase **Wall**, koristimo **for** petlju koja ponavlja izjave unutar bloka za sve **i=1,2,3,4,5** u našem slučaju. Podsjetimo se da **super** poziva konstruktor nadklase, u našem slučaju klase **World**. **Super** mora biti prvi poziv u konstruktoru.

6.8. Dodaj još više zidova

Modificirajte konstruktor klase **TestArena** tako da se ćelije postave kako je prikazano na slici.



6.9. Razmislite o tome kako predstaviti niz zidova

Razmislimo o tome koliko informacija trebamo kako bismo mogli stvoriti bilo koji niz uzastopnih zidova.

6.10. Dodaj metodu za stvaranje niza zidova

Stvorite metodu **createRowOfWalls()** u nadklasi **Arena**, koja će imati tri parametra:

- redak (gornji redak ima indeks 0) na kojem treba početi stvarati zidove,
- stupac (lijevi stupac ima indeks 0) od kojeg treba početi stvarati zidove,
- broj koji izražava koliko uzastopnih zidova treba stvoriti.

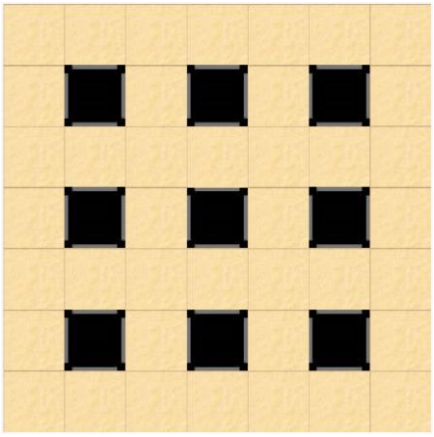
Metoda nema povratnu vrijednost (stoga koristimo ključnu riječ **void**).

6.11. Upotrijebi novu metodu

Modificirajte kôd u konstruktoru klase **TestArena** kako biste koristili metodu **createRowOfWalls()** iz njenog nadrazreda – klasa **Arena**.

6.12. Promijeni raspored zidova

Modificirajte konstruktor klase **TestArena** kako biste stvorili arenu prikazanu na slici u nastavku. Za to, modificirajte metodu **createRowOfWalls()** kako bi imala četvrti parametar koji definira razmak između zidova.



6.13. Razmislite o tome kako predstaviti pravokutnik zidova

Razmislimo o tome koliko i kakve informacije trebamo kako bismo mogli stvoriti zidove u rasporedu pravokutnika čija početna točka može biti određena i za koje se može postaviti razmak između zidova u oba reda i stupaca.

6.14. Dodaj metodu za stvaranje pravokutnika zidova

Deklarirajte metodu **createRectangleOfWalls()** u nadrazredu **Arena**, koja će imati sljedeće parametre:

- redak (gornji redak ima indeks 0) od kojeg počinju stvarati zidovi,
- stupac (lijevi stupac ima indeks 0) od kojeg počinju stvarati zidovi,
- broj redova koji će biti stvoreni,
- broj uzastopnih zidova koji će biti stvoreni u redu,
- broj praznih ćelija (redaka) između redova,
- broj praznih ćelija između zidova u redu.

Metoda nema povratnu vrijednost.

6.15. Koristi novu metodu

Modificirajte konstruktor klase **TestArena** kako biste iskoristili metodu **createRectangleOfWalls()** kako bi postavili arenu kako je prikazano u zadatku 6.12.

6.16. Testiraj svoju arenu

Stvorite nekoliko zidova u svojoj areni i dodajte dva igrača. Testirajte funkcionalnost igre, tj. provjerite hoće li se igrači ponašati ispravno, tj. neće li ući u **BrickWall** ili **Wall**.

7. Lista i for za svaku petlju

Ovo poglavlje se fokusira na liste detaljnije koristeći ih za praćenje drugih objekata u areni. Uvodi osnovne metode za rad s listom (kreiranje, dodavanje elemenata, uklanjanje elemenata, pristup elementu) i također uči kako koristiti petlju for each za jednostavan pristup svim elementima liste.

7.1. Provjeri završetak igre

Stvorite metodu **isGameEnded()** bez parametara u klasi **Arena** koja detektira je li igra završila (ostao je samo jedan ili nijedan igrač) i vraća vrijednost tipa **boolean** koja označava je li se to dogodilo. Za sada pretpostavimo da kraj igre nikada ne nastaje.

7.2. Završi igru

Dodajte metodu **act()** u klasu **Arena**. U metodi provjerite je li igra završena (koristeći metodu **isGameEnded())** i ako jest, zaustavite igru. Da biste zaustavili Greenfoot okruženje, koristite naredbu **Greenfoot.stop();**.

7.3. Dodaj listu igrača

Dodajte atribut **listOfPlayers** tipa **LinkedList<Player>** u klasu **Arena**. Ne zaboravite da morate uvesti paket s klasom **LinkedList**. Inicijalizirajte atribut u konstruktoru klase **Arena**.

7.4. Registriraj igrače

Dodajte metodu **registerPlayer()** u klasu **Arena** koja prima jedan parametar tipa **Player** i umetnite ga na kraj liste **listOfPlayers** koristeći metodu **add()**. Modificirajte podklase klase **Arena** kako biste registrirali igrača u nadklasi (**Arena**) kada se igrač umetne u svijet na odgovarajućem mjestu.

7.5. Registriraj i ukloni igrača

Dodajte metodu **unregisterAndRemovePlayer()** u klasu Arena koja prima jedan parametar tipa **Player**. Metoda uklanja igrača iz liste igrača, a zatim ga uklanja iz svijeta.

7.6. Ispravno završi igru

Implementirajte tijelo metode **isGameEnded()** tako da metoda vraća true kada u igri ostane jedan ili nijedan igrač. Koristite odgovarajuće metode liste.

7.7. Učini bombe opasnim

Modificirajte kod u metodi **act()** klase **Bomb** tako da prije nego što bomba bude uklonjena iz svijeta, uništi sve igrače koji su udaljeni najviše jednu snagu od nje. Metoda **getObjectsInRange()** vraća popis tipa **List**. Njezin prvi parametar je raspon - u ovom slučaju za promjenu snage. Njezin drugi parametar je identičan klasi čiju instancu traži unutar zadanog raspona.

7.8. Ukloni pogođene igrače

Koristite **for** petlju kako biste iterirali kroz sve igrače na listi igrača pogođenih bombom. Odregistrirajte takve igrače u klasi **Arena**.

7.9. Uredi klasu igrača

Stvorite **hit()** metodu u klasi **Player** koja će biti pozvana kada bomba igrača pogodi njega. Odregistrirajte igrača iz svijeta u ovoj metodi. Uredite kôd u metodi **act()** klase Bomb kako bi odražavao novu funkcionalnost.

7.10. Ukloni vlasnika

Stvorite metodu **removeOwner()** u klasi **Bomb** koja postavlja njen atribut **owner** na **null**. Instanca objekta može se smatrati pokazivačem - "strelicom" na objekt. Postavljanjem na **null**, pokazivač ne pokazuje ni na koji objekt.

7.11. Dodaj listu bombi

Stvorite atribut **listOfActiveBombs** tipa **LinkedList<Bomb>** u klasi **Player**. Inicijalizirajte ga u odgovarajućem konstruktoru. Modificirajte tijela metoda prema sljedećim pravilima:

- u metodi **act()** registrirajte novostvorenu bombu na **listOfActiveBombs**;

- u metodi **bombExploded()** uklonite bombu koja je došla kao parametar (ona koja je eksplodirala) iz **listOfActiveBombs**;
- u metodi **hit()** koristite petlju for each kako biste uklonili vlasnika iz svih bombi u **listOfActiveBombs**.

8. Privatne metode i while petlja

Ova tema uvodi **while** petlju - petlju koja se ponavlja dok se neka uvjet definiran na početku petlje evaluira kao **true**. Nadalje, također uči učenike kako stvoriti privatne metode - metode koje se mogu pozivati samo od instance klase u kojoj je metoda definirana.

8.1. Stvori klasu Fire

Stvorite klasu **Fire**. Odaberite odgovarajuću grafičku reprezentaciju. Konstruktor ove klase ima jedan parametar koji određuje koliko dugo vatra gori na mjestu. Osigurajte da vatra nestane iz svijeta nakon određenog vremena.

8.2. Pokrenite vatru

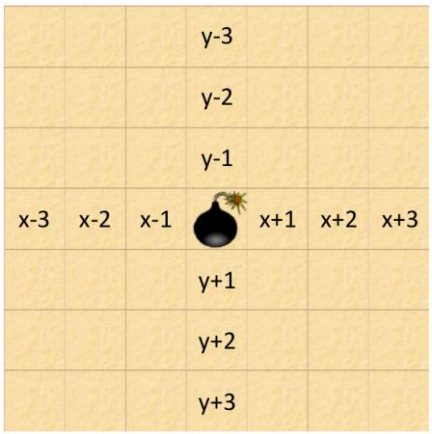
Modificirajte postojeći kôd eksplozije bombe kako bi ostavili instancu klase **Fire** na mjestu eksplozije bombe. Testirajte svoje rješenje.

8.3. Spread the fire

Koristite **for** petlju kako biste proširili eksploziju bombe (stvorili instancu klase **Fire**) u pravom smjeru od bombe. Proširite eksploziju na koliko god ćelija je naznačeno atributom snage bombe.

8.4. Spread the fire in all directions

Prilagodite eksploziju bombe tako da generira vatre u svim smjerovima. Pomozite si mijenjanjem koordinata kako je prikazano na slici u nastavku.



8.5. Prepisivanje petlji

Prepišite sve **for** petlje za širenje vatre koristeći **while** petlju. Za sada izostavite drugi dio uvjeta (moguće je staviti vatru u sljedeću ćeliju).

8.6. Koristi privatnu metodu

Koristite privatnu metodu **spreadFire()** za širenje vatre nakon eksplozije bombe.

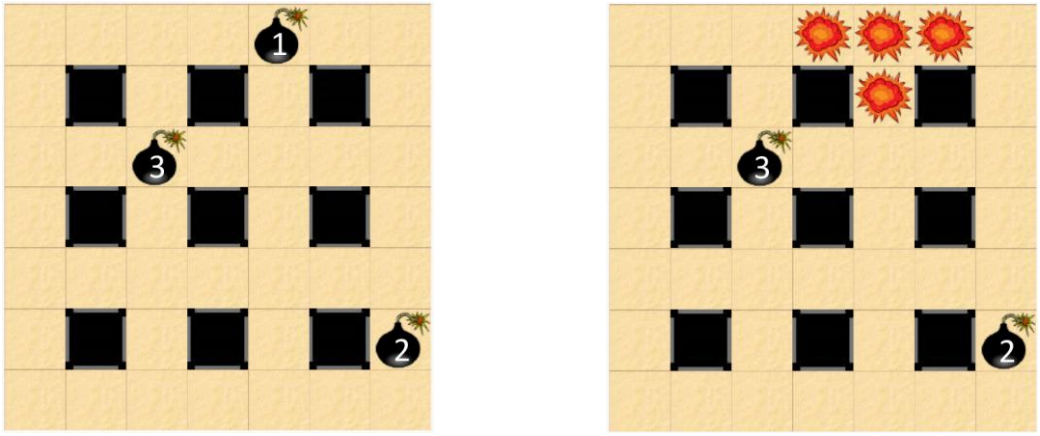
8.7. Dodaj još jednu privatnu metodu

Stvorite privatnu metodu **canCellExplode()** u klasi **Bomb** koja kao parametre uzima koordinate retka i stupca te vraća true ako eksplozija može nastati na toj ćeliji, a **false** inače. Vatra ne može nastaviti ako:

- doseže rub svijeta,
- ako ćelija sadrži zid.

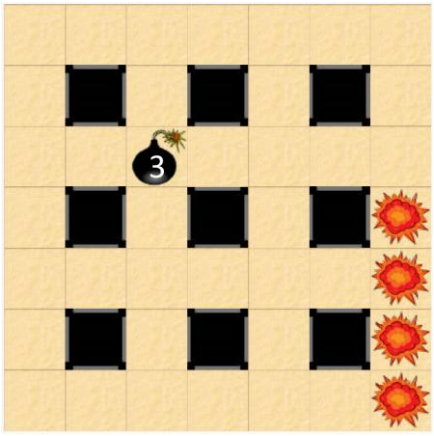
8.8. Koristi privatne metode

Koristeći metodu **canCellExplode()**, sada možete modificirati uvjet u **while** petlji u metodi **spreadFire()** u klasi **Bomb**. Promijenite uvjet u petlji tako da se poštuje rezultat provjere iz metode **canCellExplode()**. Testirajte funkcionalnost rješenja s bombama različite snage između zidova. Testovi različitih eksplozija prikazani su u sljedećim slikama:

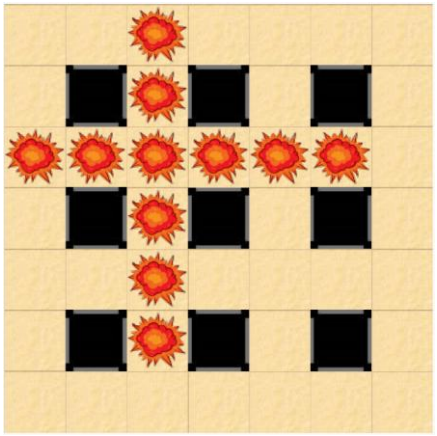


a

b



c



d

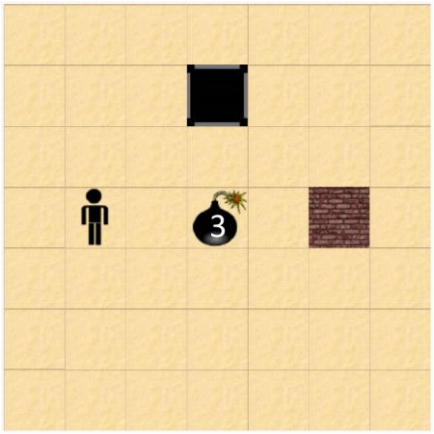
U dijelu (a) možemo vidjeti originalnu distribuciju, u dijelu (b) bomba s snagom 1 je eksplodirala, u dijelu (c) bomba s snagom 2 je eksplodirala, a u dijelu (d) bomba sa snagom 3 je eksplodirala.

8.9. Provjeri prepreku eksplozije

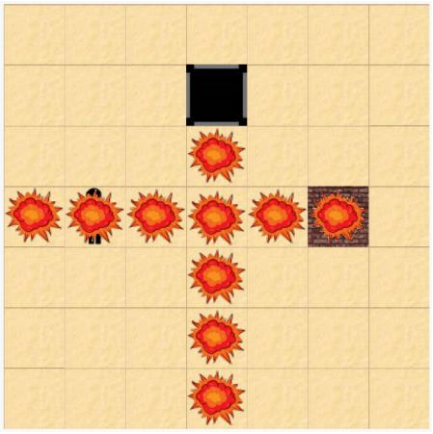
Dodaj **canExplosionContinue()** privatnu metodu klasi **Bomb** koja uzima retke i stupce kao parametre i vraća **true** ako se eksplozija nije zaustavila na danoj ćeliji. Ako se eksplozija zaustavila, metoda vraća **false**. Eksplozija ne može nastaviti ako je pogodila zid.

8.10. Limit the explosion Ograniči eksploziju

Koristeći metodu **canExplosionContinue()**, modificiraj **spreadFire()** metodu u klasi **Bomb**. Ako eksplozija može nastaviti iz dane ćelije, povećaj vrijednost varijable **i** za 1 i ponovno izračunaj koordinate nove ćelije eksplozije. Inače, umjetno povećaj vrijednost varijable **i** na vrijednost veću od snage bombe, što zaustavlja petlju. Testiraj svoje rješenje na situaciji s prikazane slike:



A



b

8.11. Modificiraj provjeru prisutnosti vatre

Modificirajte ponašanje instance klase **Bomb** tako da ne poziva metodu **hit()** igrača u svom dosegu. Umjesto toga, igrač će sam provjeriti preklapanje s vatrom. Modificiraj ponašanje metode **act()** igrača tako da prvo provjeri preklapanje s instancom klase **Fire**. Ako je preklapanje prisutno, sam će pozvati svoju metodu **hit()**. Time će se osigurati da i igrač bude pogođen vatrom koja gori nakon eksplozije bombe.

8.12. Dodaj lanac eksplozija

Modificirajte metodu **act()** klase **Bomb** tako da bomba eksplodira čak i ako se nalazi u istoj ćeliji kao i vatra. Provjeri rješenje izvodeći lančanu reakciju nekoliko bombi.

9. Polimorfizam

Cilj ovog dijela je naučiti učenike kako kreirati virtualne metode i nadopunjavati ih po potrebi u podklasama. Također, učenici će biti upoznati s novom vidljivošću atributa i metoda – **protected** pristupnim modificatorom. Učenici će koristiti polimorfizam kako bi pojednostavili postojeći kôd.

9.1. Dodaj mine

Započnimo dodavanjem mine. Mina eksplodira čim igrač na nju stupi. Mina će također uvijek eksplodirati kada je pogodi vatra (npr. od bombe koja je eksplodirala u blizini). Ostavlja vatru na svom mjestu (i samo tamo). Omogućite igraču da postavlja mine (kao bombe) kada pritisne tipku (npr. control ili shift). Slično bombama, igrač također ima ograničen broj mina (tj. ako postavi sve mine, može postaviti još jednu minu samo ako jedna od prethodno postavljenih mina eksplodira). Početni broj mina postavlja se parametrom konstruktora klase **Player**. Za registraciju mina i reakciju na njihovu eksploziju u klasi **Player**, slijedite isti postupak kao i za bombe (stvaranje liste mina, dodavanje metoda **mineExploded()**, **canPlantMine()**, itd.).

9.2. Dodaj nadklasu

Stvorite zajedničku nadklasu za klase **Bomb** i **Mine** - klasu **Explosive**. Koje atribute i metode treba premjestiti u nadklasu, a koje treba zadržati u podklasama? Modificirajte postojeće klase prema vašem dizajnu.

9.3. Prilagodi vidljivost atributa

Promijenite vidljivost atributa **owner** u klasi **Explosive** na **protected**. Vidljivost **private** atributa omogućila bi njegovo korištenje samo unutar klase **Explosive** i ne bi bila dostupna njezinim podklasama. **Protected** znači vidljivost unutar paketa, što u našem slučaju odgovara projektu **Bomberman**.

9.4. Dodaj tekstualni ispis

Stvorite metodu **printWhoYouAre()** bez parametara u klasi **Explosive** koja nema povratnu vrijednost. Metoda ispisuje tekst "EXPLOSIVE" na ekranu gdje se trenutno nalazi eksploziv. Stvorite instancu klase **Mine** i pozovite metodu **printWhoYouAre()**. Što se događa? Stvorite instancu klase **Bomb** i pozovite metodu **printWhoYouAre()**. Što se događa u tom slučaju?

9.5. Poboljšaj tekstualni opis

Stvorite metodu **printWhoYouAre()** u klasi **Mine** s istim zaglavljem kao i u klasi **Explosive** (tj. metoda će imati isti naziv, iste parametre i isti tip povratne vrijednosti). Metoda će ispisati tekst "MINE" na ekranu. Ponovno stvorite instancu klase **Mine** i klase **Bomb**. Pokušajte pretpostaviti što će se dogoditi kada pozovete testnu metodu u instanci klase **Mine** i instanci klase **Bomb**. Nakon toga, pozovite metode. Podudara li se vaša pretpostavka s rezultatom?

9.6. Dovrši tekstualni ispis

Prekrijte metodu **printWhoYouAre()** u klasi **Bomb** tako da na ekranu ispiše tekst "BOMB". Provjerite ispravnost vašeg rješenja.

9.7. Dodaj rukovanje eksplozijom

Stvorite metode **shouldExplode()** i **explosion()** u klasi **Explosive** i metodu **explosiveExploded()** u klasi **Player** kako je opisano gore. Ne implementirajte tijela metoda još. Ako je potreban povratna vrijednost, vratite **false**.

9.8. Neka eksploziv eksplodira

Napišite tijelo metode **act()** u klasi **Explosive**.

9.9. Neka bomba eksplodira

Prekrijte metode **shouldExplode()** i **explosion()** u klasi **Bomb**. Koristite odgovarajući kôd iz njezine metode **act()**. Primijetite da je moguće jednostavno napisati tijela metoda jer nema potrebe razmatrati uvjete (to je već obavila nadklasa).

9.10. Neka mina eksplodira

Prekrijte metode **shouldExplode()** i **explosion()** u klasi **Mine**. Koristite odgovarajući kôd iz njezine metode **act()**. Zašto je na kraju potrebno ukloniti metodu **act()**?

9.11. Dodaj interakciju s vatrom

Modificirajte tijelo metode **shouldExplode()** u klasi **Explosive** tako da metoda vraća **true** ako instanca dodiruje instancu klase **Fire**. Modificirajte preklopljene metode **shouldExplode()** u klasama **Bomb** i **Mine** kako biste koristili funkcionalnost nadređene metode.

9.12. Pojednostavite atribute igrača

Uklonite atribute **listOfActiveBombs** i **listOfActiveMines** iz klase **Player**. Dodajte jedini atribut **listOfActiveExplosives** tipa **LinkedList<Explosive>** u klasu **Player**. Inicijalizirajte ga u konstruktoru i uklonite inicijalizaciju originalnih atributa iz konstruktora.

9.13. Pojednostavite rukovanje eksplozijom

Implementirajte tijelo metode **explosiveExploded()**. Koristite operator **instanceof** kako biste odredili je li eksploziv bomba ili je li eksploziv mina. Na temelju njezinog stvarnog tipa, povećajte brojač dostupnih bombi ili brojač dostupnih mina. Obavezno uklonite eksploziv iz liste aktivnih eksploziva. Konačno, uklonite nepotrebne metode **bombExploded()** i **mineExploded()**.

10. Nasumični brojevi

Ova tema posvećena je nasumičnosti. Učenici će naučiti o klasi **Random**. Koristeći instance te klase, generirat će nasumične brojeve. Također, tema prikazuje način generiranja nasumičnih brojeva bez korištenja klase **Random**, izravno koristeći Greenfoot okruženje. Učenici će koristiti nasumične brojeve kako bi nasumično rasporedili izgled arene i dodali bonuse u svijet - posebne elemente koji se stvaraju nakon što zid od opeke eksplodira i poboljšavaju odabrana svojstva igrača.

10.1. Razmislite o nasumičnosti

Razmislite što je nasumičnost, kako možemo dobiti neki nasumičan rezultat iz eksperimenta i koje nasumične fenomene promatramo u svijetu oko nas.

10.2. Razmislite o generiranju nasumičnih vrijednosti

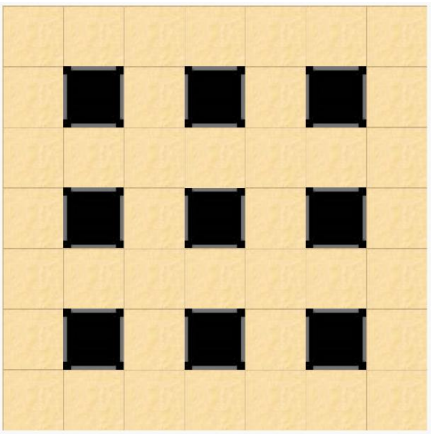
Razmotrimo klasičnu kocku s šest strana. Možemo li je koristiti za generiranje nasumične pozicije na šahovnici sa 6x6 polja? Što je sa šahovnicom s 3x3 polja? Kako bi se metoda generiranja promijenila ako bismo koristili novčić? Predložite takve algoritme generiranja pozicija.

10.3. Promatrajte nasumičnost

Koristeći algoritam iz prethodnog zadatka i uz pomoć kocke, generirajte nasumične pozicije na šahovnici. Zabilježite rezultate na šahovnici. Može li se primijetiti neka pravilnost u rezultatima?

10.4. Pripremite nasumičnu arenu

Pripremite arenu. Stvorite podklasu klase **Arena**, koju nazovite npr. **RandomArena**. Postavite odgovarajuću veličinu svijeta u konstruktoru. Preporučujemo redoviti raspored zidova s jednim praznim poljem između zidova kako je prikazano na sljedećoj slici:



10.5. Dodaj generator slučajnih brojeva

Dodaj referentni atribut tipa `Random` klasi `RandomArena`. Zapamti da je klasa `Random` definirana u paketu j `java.util.Random`. Inicijaliziraj atribut u konstruktoru.

10.6. Provjeri zauzetost ćelije

Dodaj privatnu metodu `isCellFree()` klasi `RandomArena` koja uzima dva parametra - stupac i redak. Metoda će vratiti `true` ako je ćelija slobodna u svijetu (ne sadrži instancu klase **Actor**), inače će vratiti `false`.

10.7. Generiraj slučajne zidove

Izmijenite konstruktor klase `RandomArena` kako biste slučajno generirali zidove u trećini svih ćelija na areni.

10.8. Premjesti slučajne elemente na prethodne

Premjesti metode **`createRandomWall()`** i **`isCellFree()`**, kao i atribut generator (uključujući njegovu inicijalizaciju u konstruktoru), u nadklasu **`Arena.`** Ne zaboravi premjestiti i uvozne izjave (linije).

10.9. Generaliziraj slučajne elemente

Dodaj parametar tipa **`Actor`** metodi **`createRandomWall()`** - to će biti lik kojeg ćemo umetnuti na slučajne koordinate. Promijeni naziv metode (npr. **`insertActorRandomly()`**) i ažuriraj pozivanje metode iz klase **`RandomArena.`**

10.10. Dodaj bonuse

Kreiraj klasu **`Bonus`** kao podklasu klase **`Actor`**. Kreiraj dvije podklase klase **`Bonus`** - klasu **`BonusFire`** i klasu **`BonusBomb`**. Postavi odgovarajuće slike za klase.

10.11. Stvaraj bonuse nasumično

Uredite kod u metodi **act()** klase **BrickWall**. Nakon što bude uništen, s vjerojatnošću od 10% generirajte bonus vatru, a s vjerojatnošću od 10% generirajte bonus bombu. U 80% slučajeva, nakon uništenja, ništa se ne generira.

10.12. Primijeni bonus

Pripremi klasu **`Bonus`**. Kreiraj **`protected applyYourself()`** (zaštićena metoda) bez povratne vrijednosti, koja prima jedan parametar tipa **`Player`**. Ostavi ovu metodu praznu u klasi **`Bonus`**. Zatim definiraj radnju u metodi **`act()`** kako bi prvo detektirala je li igrač stupio na bonus (metoda **`(Player)this.getOneIntersectingObject(Player.class)`**), i ako jest, primijeni ga (pozivajući metodu), te na kraju ukloni bonus iz svijeta.

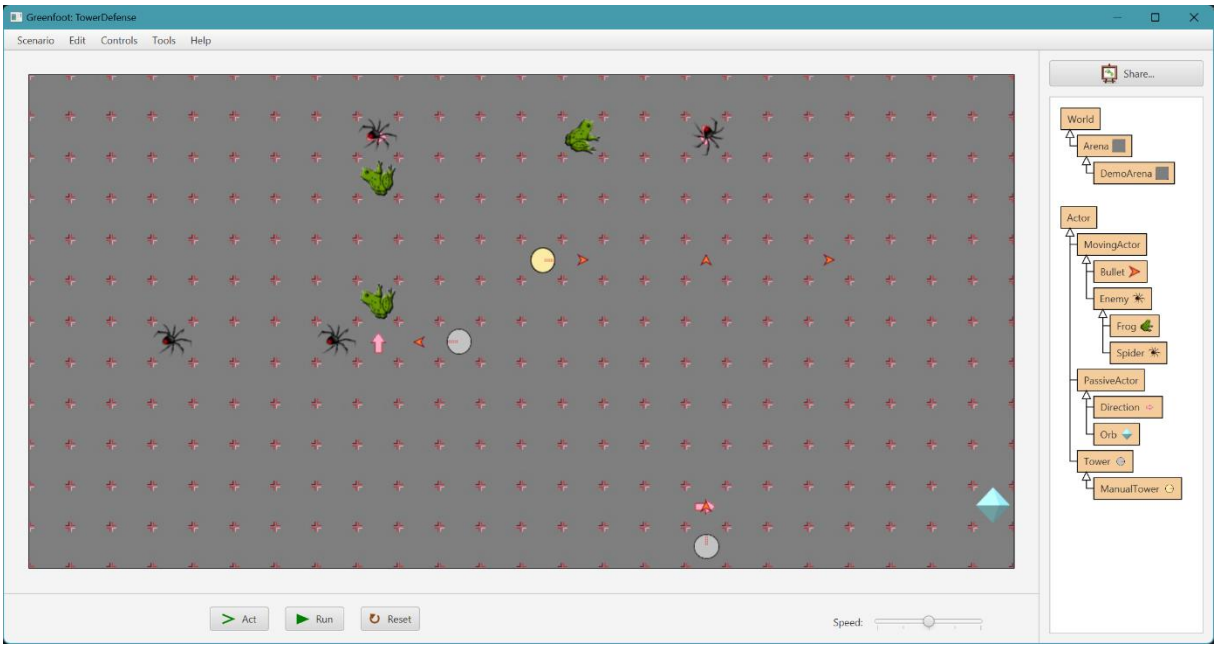
10.13. Povećaj broj bombi

Dodaj javnu metodu **`public increaseBombPower()`** bez parametara klasi **`Player`** koja nema povratnu vrijednost, i koja će povećati vrijednost atributa **`bombPower`** za jedan. Zatim prepravi metodu **`applyYourself()`** u klasi **`BonusFire`**. Primjena bonusa znači povećanje broja bombi koje igrač ima za jedan (pozivanje metode **increaseBombCount()**).

3.2. Tower defense

U igrama poput obrambene kule - Tower defense, igrač koristi tornjeve koji ispaljuju određenu vrstu metaka kako bi zaustavili neprijatelje u njihovu pohodu na uništenje kugle. Neprijatelji uvijek slijede isti put, no kako igra napreduje, postaju jači i pojavljuju se u većim skupinama. Igrač mora postaviti kule na strateška mjesta kako bi ih zaustavio u nadolazećim valovima. Postoje mnoge verzije igre koje se odvijaju u različitim svjetovima koristeći različite entitete (od balona do orkova), braneći se koristeći tornjeve nalik životinjama pa sve do čarobnih bazena.

Ovaj projekt će predstaviti jednu vrstu Tower defense igre tj. obrambene kule koju igrač može ili ne mora ručno kontrolirati. Neprijatelji će biti različitih vrsta s različitim vrijednostima zdravlja (HP) i brzinama. Predstavljeni dizajn igre je lako proširiv, što ostavlja dovoljno prostora za kreativnost studenata kao i za zadatke nastavnika. Iz tog razloga, nastojali smo minimizirati aktivnosti koje se fokusiraju na ocjenjivanje u prvim poglavljima. Nadalje, dizajn ostavlja dovoljno prostora za prirodno uvođenje tema izvan dosega osnovnog objektno orijentiranog programiranja (OOP), kao što je polimorfizam, s lakoćom. Projekt u svom konačnom stanju tijekom igranja prikazan je na Slici 3.



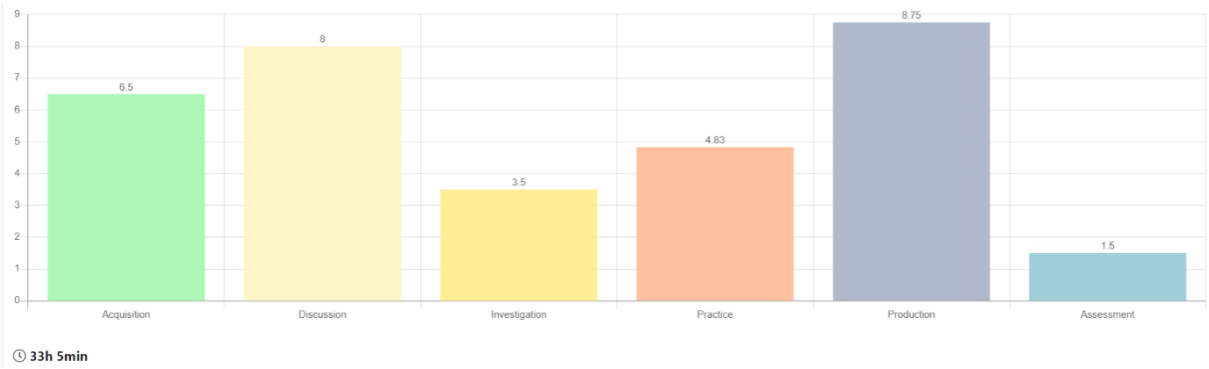
Slika 3: Greenfoot okruženje s konačnim stanjem projekta Tower defense

Izvorni kodovi dostupni su na:
https://gitlab.kicon.fri.uniza.sk/oop4fun/project-tower-defense

Dizajn učenja dostupan je na:
http://learning-design.eu/en/preview/452257b563cbf14b6f06acfd/details

3.2.1. Sadržaj i opseg obrazovnog programa

Ukupno radno opterećenje učenika je 33 sata i 5 minuta i raspoređeno je na sljedeći način:



Slika 4: Opterećenje učenika prilikom korištenja projekta Tower defense

Konstruktivno usklađivanje sažeto je u tablici u nastavku:

Tablica 3. Konstruktivno usklađenje projekta 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						

Detaljan plan pogledati u prilogu 5.2.

3.2.2. Teme

Projekt Tower defense je podijeljen u sedam tema:

Pokrivene teme osnovnog (laganog) OOP-a uključuju:

- klase, objekte, instance
- metode, proslijeđivanje argumenata metodi
- konstruktore
- atribute
- statičke varijable i metode
- enkapsulaciju
- naslijeđivanje
- apstraktne klase
- životni ciklus objekta

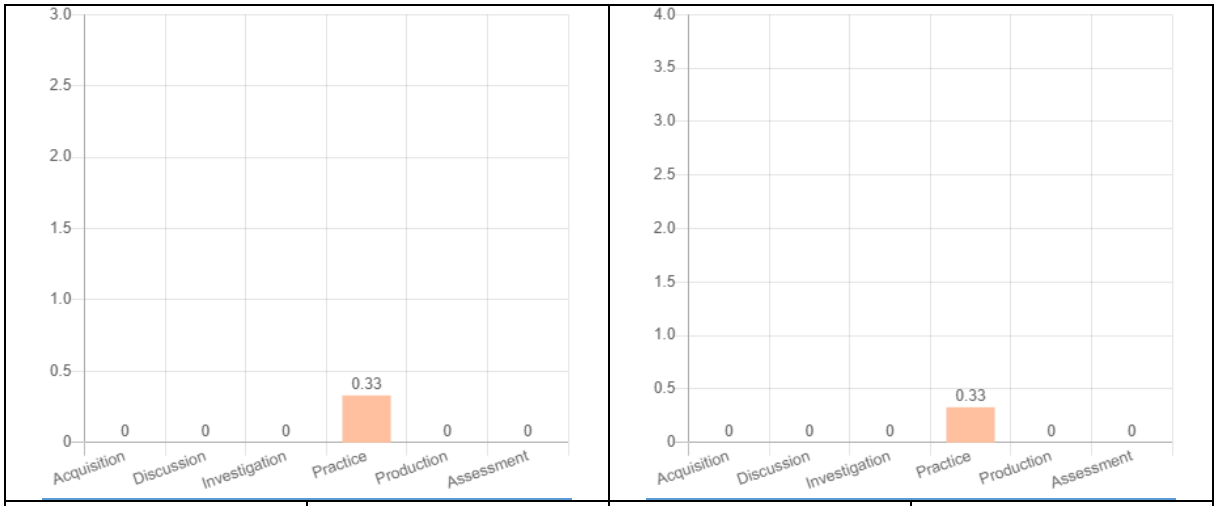
1. Uvod u Greenfoot okruženje

Tema je posvećena stvaranju projekta. Učenici će biti sposobni stvoriti novi projekt u Greenfoot okruženju, kreirati klasu (kao podklasu klase **`Actor`**), odabrati sliku za novokreiranu klasu, stvoriti njezinu instancu i poslati joj poruku.

Stvorite novi projekt. Dodijelite mu odgovarajuće ime (npr. **`TowerDefense`**) i spremite ga na odgovarajuće mjesto.

Tablica 4 sumira usporedbu opterećenja teme Greenfoot okruženja između projekata Bomberman i Obrambena kula. Ne postoji razlika u dizajnu tema.

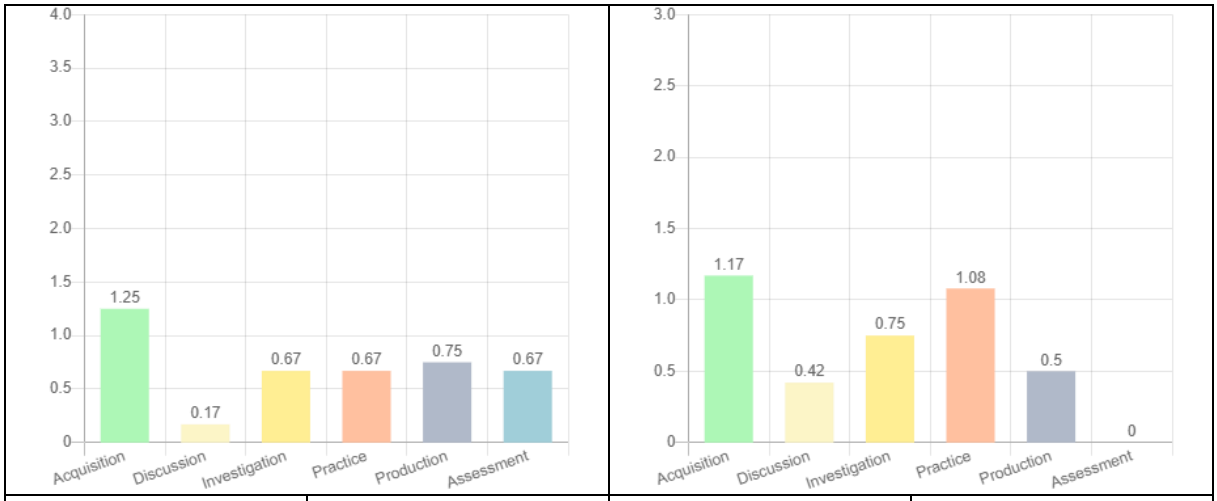
Tablica 4. Usporedba opterećenja teme Greenfoot okruženje između projekata Bomberman i Tower defense



Bomberman	0h 20min	Tower defense	0h 20min

Tablica 5 sažima usporedbu opterećenja teme Definicija klase između projekata Bomberman i Tower Defense. Opterećenje projekta Tower defense je niže, više praktično orijentirano s naglaskom na istraživanje i praksu.

Tablica 5. Usporedba opterećenja teme Definicija klase između projekata Bomberman i Tower Defense



Bomberman	4h 10min	Tower defense	3h 55min

1.1. Identificiraj objekte iz projekta Bomberman

1.2. Pripremi svijet

Uredi izvorni kôd klase **`MyWorld`** (dvostruko kliknite na nju) kako bi stvorili svijet veličine 24x12 ćelija. Svaka ćelija treba biti veličine 50 piksela.

1.3. Pripremi grafiku svijeta

Pronađi ili stvori odgovarajuću sliku za pozadinu svijeta. Možeš koristiti već pripremljene slike (odaberi stavku `Set image...` iz kontekstnog izbornika klase `MyWorld`) ili prilagođenu sliku (kopiraj sliku u podmapu `images` u svojoj projekt mapi i odaberi je koristeći isti postupak kao što je opisano ranije).

Kao pozadinu možeš koristiti jednu sliku koja će pokriti cijelu površinu svijeta (izračunaj potrebnu veličinu slike s obzirom na veličinu svijeta) ili manju sliku koja će biti ponovno kopirana (koristi kvadratnu sliku veličine ćelije).

1.4. Create class Enemy Stvori klasu Enemy (neprijatelj)

Stvori neprijatelja. Neprijatelj će se kretati prema igračevoj kuli kako bi je oštetio i eventualno uništio. Stvori novu podklasu klase **Actor** (odaberi stavku `New subclass...` iz kontekstnog izbornika klase `Actor`). Daj joj odgovarajuće ime (`Enemy`) i sliku.

1.5. Stvori instancu klase Enemy

Stvori instancu klase Neprijatelj (odaberi stavku `new Enemy()` iz kontekstnog izbornika klase class `Enemy`, stavi instancu u svijet lijevim klikom miša na željenu poziciju). Istraži njeno unutarnje stanje (odaberi stavku `Inspect` iz kontekstnog izbornika stvorene instance).

Stvori još jednu instancu klase **`class Enemy`** i stavi je na drugu poziciju. Usporedi unutarnja stanja dviju stvorenih instanci

1.6. Pošalji poruku instanci

Pošalji poruku instanci klase `Enemy` (odaberi stavku **`inherited from Actor`** iz kontekstnog izbornika odabrane instance, a zatim odaberi željenu stavku). Što se dogodilo? Kako je unutarnje stanje odgovarajuće instance utjecalo?

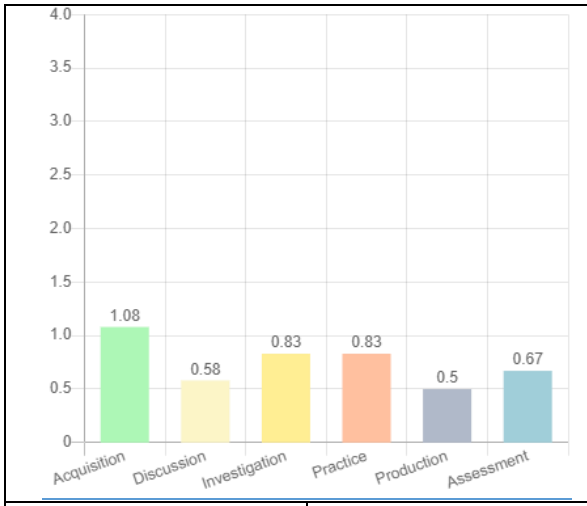
Pošalji poruke instanci klase `Enemy` tako da se pomakne na poziciju [12, 6] i okrenuta prema dolje. Zapiši slijed poslanih poruka na papir.

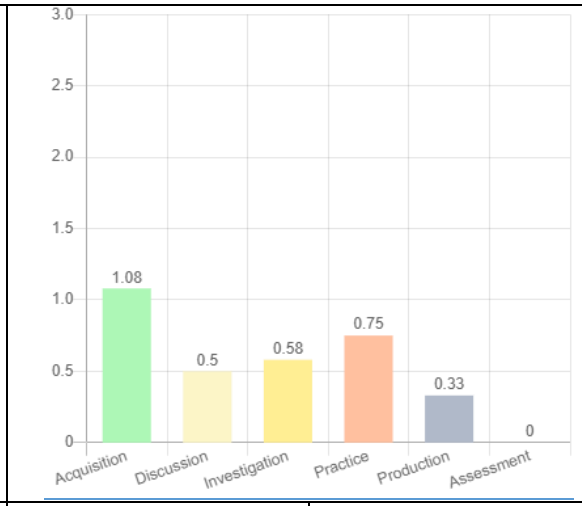
2. Algoritam, kontrole aplikacije, kreiranje metode

Tema se bavi osnovama algoritmiranja i uvodi rad s dokumentacijom od samog početka. Učenici će biti sposobni pozivati metodu u izvornom kôdu, pisati i pozivati dokumentaciju.

Tablica 6 sažima usporedbu opterećenja teme Algoritam između projekata Bomberman i Tower defense. Dizajn Tower defense sličan je Bombermanu, ali s značajno manjim brojem TLA istraživačkog tipa. Primijetite da su mnogi TLA-ovi isti kao i u projektu Bomberman. To je zato što u ovoj fazi projekata postoji veliko preklapanje onoga što se može učiniti s njima. Moguće je pronaći inspiraciju u zadacima u projektu Bomberman, ako će biti potrebno ojačati istraživački dio nastavnog plana. Međutim, radi lakšeg poučavanja OOP-a korištenjem projekta Tower defense, smatramo da je predložena količina TLA istraživačkog tipa dovoljna.

Tablica 6. Usporedba opterećenja teme Algoritam između projekata Bomberman i Tower defense





Bomberman	4h 30min	Tower defense	3h 15min

2.1. Zadatak 2.1 Napišite jednostavan algoritam iz projekta Bomberman.

2.2. Zadatak 2.2 Napišite općenitiji algoritam iz projekta Bomberman.

2.3. Pozovite metodu

Dodajte izjavu u tijelo metode **`act()`** tako da se instanca klase **`Enemy`** pomiče dvije ćelije u trenutnom smjeru kretanja. Zatim stvorite više instanci klase Neprijatelj i pozovite metodu na svakoj instanci. Je li ponašanje očekivano?

2.4. Dodavanje dokumentacije

Dodajte komentar za dokumentaciju za metodu **`act()`**

2.5. Dodavanje dodatne dokumentacije

Uredi komentar dokumentacije klase **`Enemy`**. Dodaj verziju klase i njenog autora.

2.6. Zadatak 2.7 Pročitajte dokumentaciju iz projekta Bomberman

2.7. Zadatak 2.8 Istražite kontrole aplikacija iz projekta Bomberman

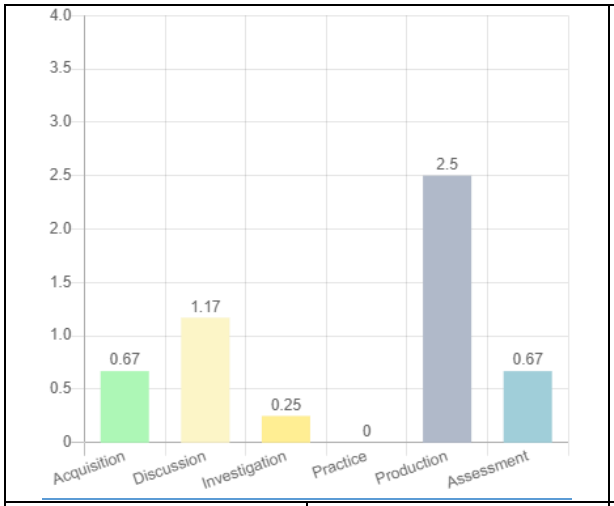
3. Grananje i kontrola neprijatelja

Tema obuhvaća nepotpuno i potpuno grananje. Uvod u osnove percepcije **`Actor`**'s **`World`** je objašnjeno. Učenici će biti sposobni pisati kod koristeći uvjete.

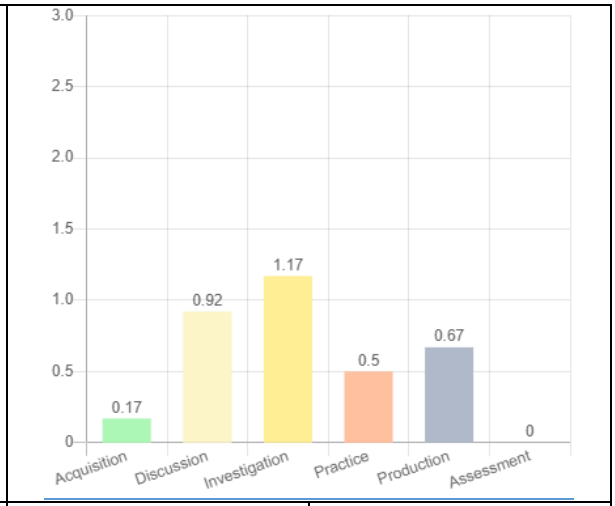
Stanje projekta u ovom poglavlju otvara mogućnosti za nastavnika da dodijeli zadatke poput "koristi instance klase **`Direction`** i **`Orb`** kako bi navigirao **`Enemy`** (neprijatelja) tako da će se kretati u željenom obrascu", s uvjetima ograničenja poput korištenja maksimalnog broja instanci određene klase ili zadatke poput "predvidi kretanje u željenom postavu svijeta".

Tablica 7 sažima usporedbu radnih opterećenja teme Grananje između projekata Bomberman i Tower defense. Postoji jasna razlika između dizajna. Tower defense razdvaja tipove produkcije TLAs kako bi se ojačalo istraživanje i primjena. To omogućuje više eksperimentiranja i otvara vrata kreativnosti studenata. Primijetite nizak broj TLAs za područje akvizicije. To je zato što a) nije uvedeno višestruko grananje i b) naglašen je istraživački tip/fokus uključenih TLAs.

Tablica 7. Usporedba opterećenja teme Algoritam između projekata Bomberman i Tower defense



Bomberman	5h 15min



Tower defense	3h 25min

3.1. Promatraj stanje neprijatelja

Stvori instancu klase **`Enemy`** i postavi je u središte ploče. Otvori prozor s unutarnjim stanjem instance i pozicioniraj ga tako da bude vidljiv dok aplikacija radi. Zatim pokreni aplikaciju i promatraj kako se vrijednosti **`x`**, **`y`** i **`rotation`** (rotacija) atributa u klasi **`Enemy`** mijenjaju. Kako se te vrijednosti mijenjaju dok se kreće (prema gore, dolje, lijevo i desno) i okreće?

3.2. Dodajte otkrivanje rubova svijeta

Dodajte kod u tijelo metode **`act()`** kako biste rotirali neprijatelja za 180° nakon što dosegnete rub svijeta.

3.3. Dodajte klase Direction and Orb

Kreirajte dvije nove klase, nasljednice klase **`Actor.`** Prva klasa bit će klasa **`Direction`** (Smjer), a druga klasa bit će **`Orb`** (Kugla). Pripremite odgovarajuće slike (maksimalno 50x50 piksela) u grafičkom uređivaču. Zatim dodijelite te slike novo stvorenim klasama.

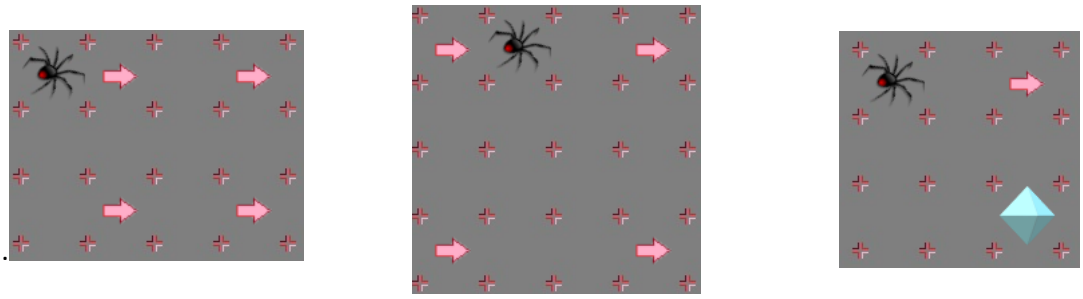
3.4. Dodaj detekciju sudara

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

- neprijatelj se okrene za 90° u smjeru kazaljke na satu kada uđe u ćeliju koja sadrži instancu klase **`Direction`**,
- neprijatelj se okrene za 90° u smjeru suprotnom od kazaljke na satu kada uđe u ćeliju koja sadrži instancu klase **`Orb`**.

3.5. Predviđanje kretanja neprijatelja na prilagođenom postavu

Pripremi različite konfiguracije, inspiraciju možeš pronaći na slikama u nastavku. Pretpostavi kako će se neprijatelj kretati? Pokreni aplikaciju. Odgovara li tvoje predviđanje onome što promatraš? Što je uzrokovalo razlike između predviđanja i stvarnosti?



a) b) c)

Slika 5: Konfiguracije prilagođenih postava primjeraka za predviđanje kretanja primjerka klase Enemy (Neprijatelj)

3.6. Predviđanje kretanja neprijatelja na specifičnom postavu

Pripremi situaciju kako je prikazano na slici ispod. Pretpostavljaš kako će se neprijatelj kretati? Pokreni aplikaciju. Odgovara li tvoje predviđanje onome što promatraš? Što je uzrokovalo razlike između predviđanja i stvarnosti?

a) b) c)

Slika 6: Konfiguracije lukavih postava instanci za predviđanje kretanja instanci klase Enemy (Neprijatelj)

3.7. Koristite potpuno grananje s detekcijom sudara

Iz posljednjeg poglavlja može se vidjeti da se problem javlja kada je instanca klase **Orb** ili **Direction** na rubu svijeta, ili kada su obje instance u istoj ćeliji. Kada grananje nije potpuno, dolazi do sudara ili ponovljenih rotacija. U osnovi, više uvjeta je zadovoljeno u isto vrijeme. Stoga, trebate promijeniti kod metode **act()** klase **Enemy** kako bi se omogućila samo jedna rotacija. Potrebno je koristiti potpuno grananje. Kreirajte kaskadu uvjeta. Najvažnija provjera (tj. prva) je provjera ruba. Druga najvažnija provjera je provjera dodira s instancom klase **Direction**. Posljednja je provjera dodira s instancom klase **Orb**. Modificirajte metodu **act()** prema tim pravilima.

3.8. Predvitie kretanje neprijatelja na temelju prethodnih poteza

Prođite iznova kroz zadatke **Error! Reference source not found.** i 3.6. Što se promijenilo?

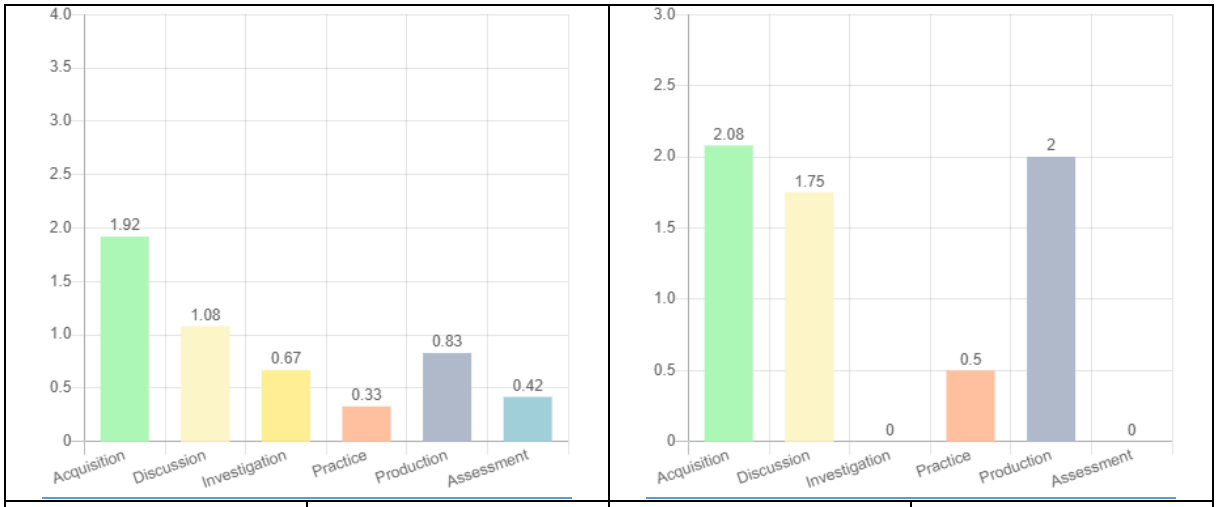
4. Varijable i izrazi

Ovo poglavlje razmatra varijable i izraze.

Stanje projekta u ovom poglavlju otvara slične mogućnosti kao i u prethodnom poglavlju. S malo kreativnosti mogu se dodati dodatne klase poput klase **`Direction`** (Smjer), koje će izazvati različito ponašanje kada na njih stupi primjerak klase **`Enemy`** (Neprijatelj) (npr. teleportacija, tuneli, itd.). Ove klase mogu se raspravljati s učenicima, a odgovarajuća implementacija može im se dodijeliti kao zadatak za domaću zadaću.

Tablica 8 sažima usporedbu radnog opterećenja tema Varijable i izrazi između projekata Bomberman i Tower defense. Pogledajte sličnost između dizajna prethodnih i trenutnih tema između ta dva projekta. Ova tema je više usmjerena na proizvodnju (slično kao i prethodna tema u Bombermanu), i obrnuto (imajte na umu da je Bomberman drugačiji tip projekta, gdje je kreativnost učenika u posebim slučajevima korištena u ovoj temi).

Tablica 8. Usporedba opterećenja tema Varijable i izrazi između projekata Bomberman i Tower defense



Bomberman	5h 15min	Tower defense	6h 20min

4.1. Promijeni smjer

Promijenite kod u metodi **`Enemy`**'s **`act()`** tako da će se okrenuti u istom smjeru kao instanca klase **`Direction`** (imati će istu rotaciju). Koristite metodu **`getOneIntersectingObject(_cls_)`** kako biste spremili instancu u odgovarajuću lokalnu varijablu (**`Direction direction`** - trebat ćete koristiti tipiziranje budući da je povratna vrijednost tipa **`Actor`**, napišite **`(Direction)`** ispred poziva metode **`getOneIntersectingObject`**, doći ćemo do tipiziranja kasnije). Ako je dobivena instance odgovarajuće klase, izdvojite rotaciju **`direction`**'s koristeći metodu **`getDirection()`** (pohranite je ako želite) te je zatim postavite na neprijatelja (**`this`**) koristeći metodu **`setDirection(int)`**. Testirajte svoje rješenje.

4.2. Preimenujte klasu MyWorld u Arena

Dajte klasi **MyWorld** bolje ime. Preimenuj ga u **Arena**. Ne zaboravite preimenovati konstruktor u skladu s tim.

4.3. Izradi plan Arene

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:

Izradi prilagođeni raspored Arene. Ispuni konstruktor klase. Dodaj jedanu instancu **Enemy**, jednu instancu **Orba** i barem jednu instancu **Direction**. Za dodavanje subclass of) **Actor**, možete koristiti sljedeći predložak:

1. Deklarirajte i inicijalizirajte varijablu potrebnog tipa (podklasa Actora)
   `Enemy e = new Enemy();`
2. Postavite svojstva koristeći odgovarajuće metode
   `e.setRotation(90);`
3. Postavite ga u svijet (**Arena**) koristeći metodu **`addObject(Actor)`**.
   `this.addObject(e, 6, 0);`

Provjerite vaše rješenje.

4.4. Identificirajte problem s kretanjem i predložite rješenje

Utvrdite što uzrokuje probleme s kretanjem. Kako se ti problemi mogu riješiti?

Enemy se trenutno pokreće 2 ćelije odjednom, što uzrokuje probleme s kretanjem. Možemo modelirati brzinu neprijatelja na drugačiji način. Instanca klase **Enemy** će uvijek pomicati samo 1 ćeliju odjednom. Međutim, uvođenjem kašnjenja u kretanju **delay** – instanca klase **Enemy** će se pomicati nakon što prođe određeni broj poziva metode **`act()`**

4.5. Attribut Enemy.moveDelay

Dodajte novi atribut tipa i **`int`** nazvan **moveDelay** u klasu **Enemy** Stvorite parametarski konstruktor s parametrom kako biste inicijalizirali ovaj atribut. Inicijalizirajte atribut s parametrom. Prilagodite kod u klasi **Arena** prema tome.

4.6. Kretanje neprijatelja uz poštivanje kašnjenja

Ažurirajte metodu **`act()`** klase **Enemy**, tako da se pomiče nakon **moveDelay** poziva metode. Uvedite novi atribut **nextMoveCounter**. Inicijalizirajte ga na **0**. Promijenite metodu **`act()`** tako da poziva **`this.move(1`**)samo ako if **nextMoveCounter** dosegne 0. Nakon kretanja, resetirajte **nextMoveCounter** na vrijednost **moveDelay**. Ako instanca klase **Enemy** nije mogla napraviti pomak (jer **nextMoveCounter** nije dosegnuo 0), smanjite n **nextMoveCounter** za 1.

4.7. Parametrijski konstruktor klase Direction (Smjer)

Dodaj parametrijski konstruktor klasi **Direction** (Smjer) s jednim parametrom **rotation** tipa int. Rotiraj stvorenu instancu u tijelu konstruktora prema parametru. Prilagodi kod u **Arena** prema tome.

4.8. Preoptereti konstruktore u klasi Direction

Preoptereti konstruktore u klasi **Direction** dodavanjem konstruktora bez parametara. U tijelu konstruktora bez parametara, pozovi parametarski konstruktor s argumentom **direction** jednakim 0.

Prilagodi kod u **`Arena`** prema tome - gdje je moguće, pozovi verziju konstruktora klase **`Direction`** bez parametara.

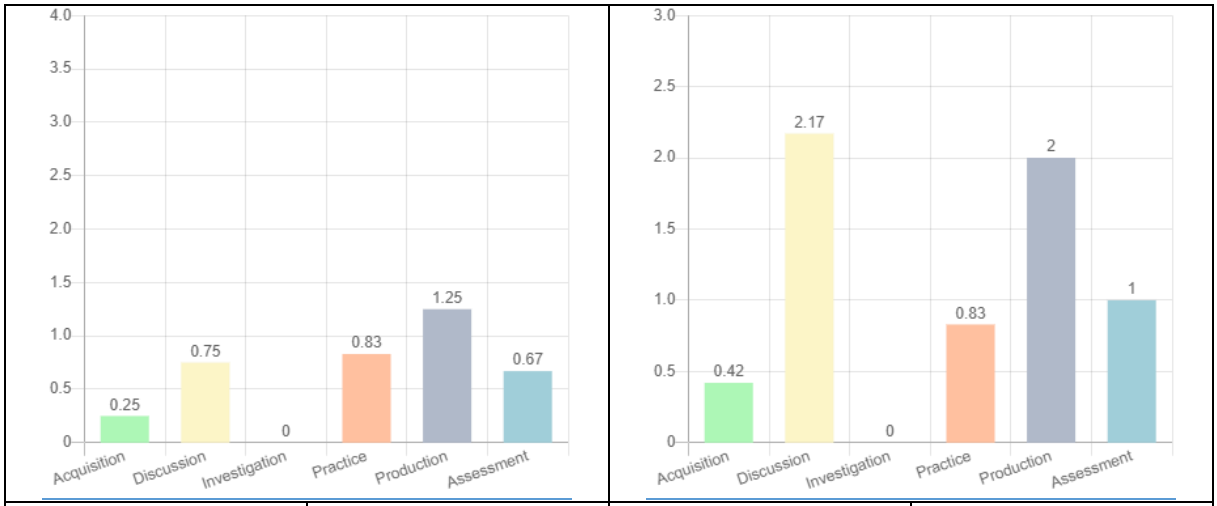
5. Povezanost

Najvažnija tema ovog projekta je povezanost. Rasprava se koristi kako bi se otkrilo kako suradnja (povezanost) različitih objekata može donijeti složeno ponašanje, iako su kodovi u objektima jednostavni za razumijevanje i upravljanje. UML dijagrami sekvenci koriste se za ilustriranje povezanosti objekata i širenje algoritma među kooperativnim objektima. Takav dijagram mote izraziti tijekom diskusije u razredu s učenicima.

Projekt se može smatrati dovršenim nakon ove teme. Sljedeća poglavlja uvode više varijabilnosti u aplikaciju s fokusom na prednosti OOP-a kada se pravilno koristi.

Tablica 9 sažima usporedbu opterećenja teme Povezanost između projekata Bomberman i Tower defense. Smatramo da je razumijevanje povezanosti najvažnijom kompetencijom prilikom korištenja ovog projekta za poučavanje OOP-a. Stoga smo značajno ojačali TLAs za produkciju i raspravu. Napomenimo da postoje i procjene u TLAs. One su dizajnirane tako da koriste prethodno obavljene TLAs u nešto drugačijem kontekstu.

Tablica 9. Usporedba opterećenja teme Povezivanje između projekata Bomberman i Tower defense



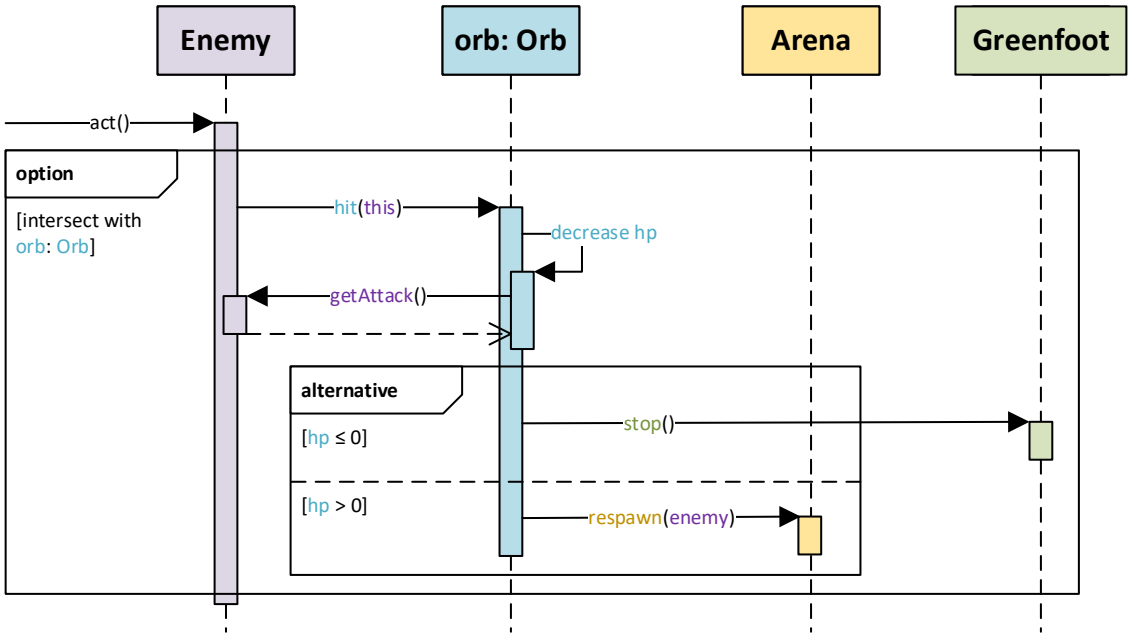
Bomberman	3h 45min	Tower defense	6h 25min

5.1. Raspravite što bi se trebalo dogoditi kada neprijatelj dođe do kule

Nakon što neprijatelj dođe do kule, HP orbite se smanjuje. Ako je HP = 0, igra završava, inače se neprijatelj ponovno pojavljuje u areni.

5.2. *Raspravite kako će instanca klase Enemy komuniicirati s relevantnim objektima putem poruka kada udari u instancu klase Orb*

Algoritam je raspršen među surađujućim objektima.



Slika 7: UML dijagram sekvenci instanci klase Enemy koji ikomuniciraju s drugim objektima kada udare u instancu klase Orb

5.3. *Atributi Enemy.attack i Orb.hp*

Dodaj novi atribut tipa **`int`** nazvan **`attack`** u klasu **`Enemy`**. Dodaj parameter u konstruktor za inicijalizaciju ovog atributa. Inicijaliziraj atribut s parametrom.

Dodaj novi atribut tipa **`int`** nazvan **`hp`** u klasu **`Orb`**. Dodaj paremetarski konstruktor s parametrom za inicijalizaciju ovog atributa. Inicijaliziraj atribut s parametrom.

Prilagodi kod u **`Arena`** prema tome.

5.4. *Getter atributa Enemy.attack*

Stvori getter atributa **`attack`** u klasi **`Enemy`**. Getter je specifična metoda korištena za dohvat vrijednosti atributa.

5.5. *Stvorite i testirajte metodu Arena.respawn(Enemy)*

Dodajte metodu **`respawn`** bez povratne vrijednosti i s jedinim parametrom tipa **`Enemy`** u klasu **`Arena`**. U metodi, postavite lokaciju i rotaciju **`enemy`** na iste vrijednosti kao i prilikom stvaranja u konstruktoru.

Testirajte metodu. Nakon što je stvorena instanca **`Arena`** (potrebno je desnom tipkom miša kliknuti na instancu arene gdje ne postoji instanca druge klase), ne pokrećite aplikaciju. Umjesto toga, povucite instancu **`Enemy`** (Neprijatelja). Zatim pozovite kontekstni izbornik instance **`Arena`** i odaberite stavku metode **`respawn`**. Kako biste popunili parametar, provjerite je li aplikacija pauzirana i je li polje s

parametrom aktivno (s treptajućim kursorom unutar njega). Ako je tako, lijevim klikom odaberite instancu **`Enemy.`** Promatrajte koja je izražena naredba izgrađena u prozoru. Zatim kliknite gumb OK i vidite što se događa.

5.6. Stvorite i testirajte metodu Orb.hit(Enemy)

Dodajte metodu **`hit`** bez povratne vrijednosti i s jedinim parametrom tipa **`Enemy`** u klasu **`Orb`**. Ostavite tijelo metode prazno.

Testirajte poziv metode. Koristite slične korake kao gore, međutim, pozovite kontekstni izbornik instance klase **`Orb`**. Promatrajte koji je izraz izgrađen u prozoru.

5.7. Pozovite metodu Orb.hit(Enemy) iz klase Enemy

Promijenite kod u metodi **`act()`** klase **`Enemy`** tako da će se metoda **hit()** pozvati kada instanca N **`Enemy`** (**`this`**) udari u instancu **`Orb`**.

Uklonite stari kod koji je uzrokovao rotaciju neprijatelja kada je kugla pogođena, kao i kod koji je uzrokovao odbijanje neprijatelja od ruba svijeta.

5.8. Implementirajte tijelo metode Orb.hit (Enemy)

Implementirajte tijelo metode **`Orb.hit(Enemy)`** s obzirom na analizu provedenu u zadatku 5.2. Testirajte svoju aplikaciju.

5.9. Dodajte klase Bullet and Tower

Dodajte klase **`Bullet`** and **`Tower`**. Koristite iste principe kao u zadatku 3.3.

5.10. Raspravite kako bi se instanca klase Bullet (Metak) trebala kretati i što bi se trebalo dogoditi kada dosegne instancu klase Enemy ili rub arene.

Metak bi se trebao kretati dok ne dosegne neprijatelja ili rub svijeta. Metak ne mijenja smjer kretanja. Brzinom metka može se upravljati koristeći isti mehanizam kao u zadatku ***Error! Reference source not found.****.*

5.11. Implementiranje kretanja instance klase Bullet

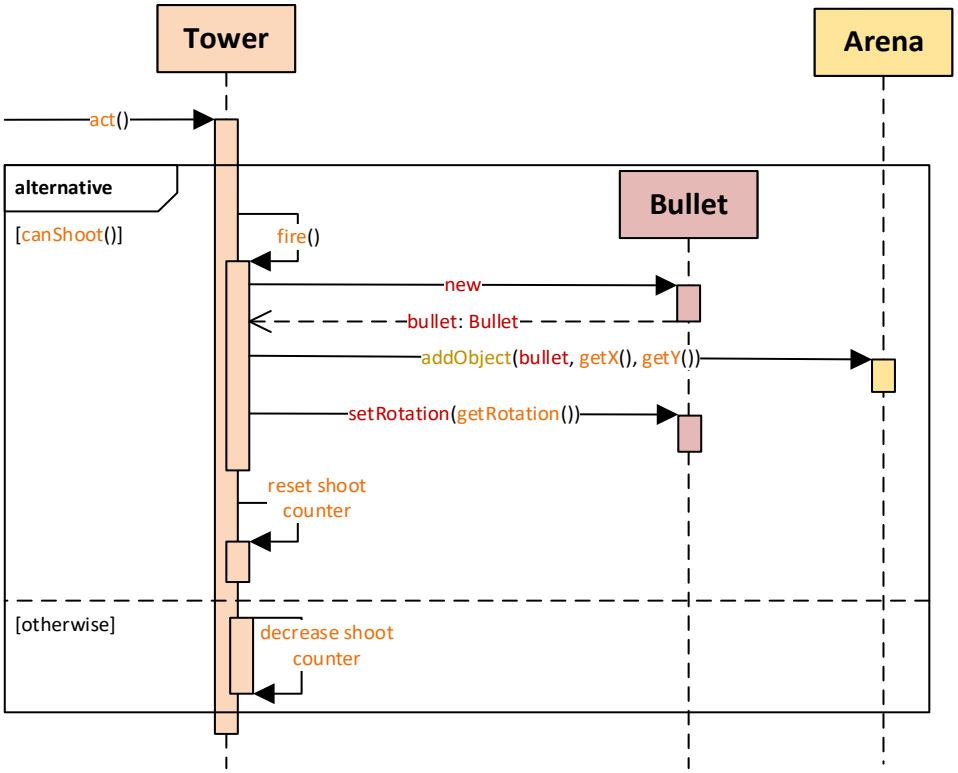
Primijenite znanje obrađeno u zadacima **Error! Reference source not found.**, 3.4 i **Error! Reference source not found.** Stavite dokumentacijski komentar u kod, gdje bi trebalo doći do interakcije s instancom klase **`Enemy.`**

5.12. Raspravite kako će instanca klase Tower pucati instancom klase Bullet

Imajte na umu da instanca **`Tower`** ne bi trebala pucati u svakom pozivu metode **`act()`**. Inspirisajte se mehanizmom korištenim u zadatku 4.6. Razdvojite relevantne korake u metode klase **`Tower`**.

5.13. Raspravite kako bi instanca klase Kula trebala interagirati s relevantnim objektima koristeći poruke prilikom pucanja

Koristite analogne principe kao u zadatku ***Error! Reference source not found.****.*



Slika 3: UML dijagram sekvenci instance klase Tower koja komunicira s drugim objektima prilikom stvaranja instanci klase Bullet

5.14. Implementiranje pucanja instance klase Tower

Slijedite rezultate zadatka 5.13:

- Prvo pripremite potrebne atribute i zatim konstruktor
- kreirajte metode **`boolean Tower.canShoot()`** i **`void Tower.fire()`** (prva neka vrati false, druga neka ne radi ništa, tako da ih možete koristiti u metodi **`act()`**), a zatim
- Implementirajte tijelo metode **`act()`**.
- Implementirajte metodu **`canShoot()`** da **`return true`** ako brojač udaraca dosegne 0.

Implementirajte metodu **`fire()`** na sljedeći način:

- pozvati konstruktor klase **`Bullet`** i pohraniti kreiranu instancu u lokalnu varijablu (**`Bullet bullet`**),
- dodajte stvoreni **`bullet`** u arenu na istim koordinatama kao instanca klase **`class Tower`** (**`this`**),
- Postavite **`bullet`** na istu rotaciju kao streljački toranj.

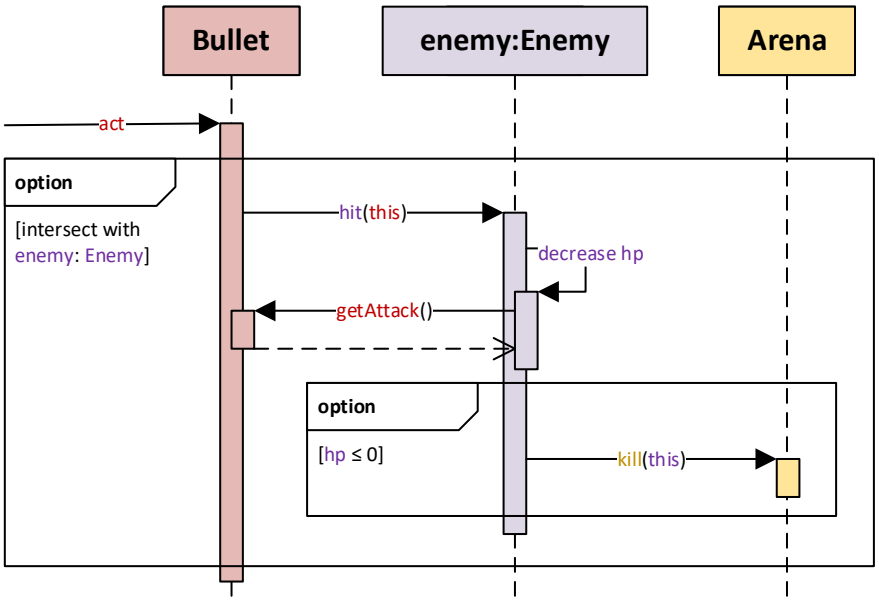
Testirajte svoje rješenje.

5.15. Towers in Arena (Tornjevi u Areni)

Preoptereti konstruktor klase **`Tower`** tako da prihvaća parametar i **`int rotation`** (analogno 4.8). Ažurirajte konstruktor klase **`Arena`** za postavljanje instanci klase **`Tower`** po želji. Koristite odgovarajući konstruktor klase **`Tower.`**

5.16. *Raspravite o tome kako bi instanca klase Bullet trebala komunicirati s relevantnim objektima pomoću poruka*

Koristi analogne principe kao u ***Error! Reference source not found.****.*



Slika 9: UML sekvencijski dijagram instance klase Bullet u interakciji s drugim objektima prilikom pogađanja instance klase Enemy

5.17. *Implementirajte instancu klase Bullet koji pogađa instancu klase Enemy*

Slijedite rješenje zadatka 5.16:

- prvo pripremite atribute i metode (analogno zadacima **Error! Reference source not found.**, **Error! Reference source not found.**, **Error! Reference source not found.** i **Error! Reference source not found.**),
- Zatim pozovite metodz **`Enemy.hit(Bullet)`** iz instance klase **`Bullet`** (analogno **Error! Reference source not found.**) gdje je ostavljen komentar u **Error! Reference source not found.** i,
- posljednje, implementirate metodu **`Enemy.hit(Bullet)`** (analogno **Error! Reference source not found.**).

Testirajte svoje rješenje.

5.18. *Pojava neprijatelja i kraj igre*

Koristite metodu **`Arena.act()`** za pozivanje stvaranja neprijatelja. Ispravno implementirajte odgodu između pojavljivanja neprijatelja. Proces mriještenja (kreirajte instancu klase **`class Enemy`**, dodijelite njena svojstva, dodajte je u arenu) implementirajte u metodi **`Arena.spawn()`**. Broji stvorene instance klase **`class Enemy`** u atributu klase c**lass Arena** (inicijalizirano na 0, povećava se kada se pojavi, smanjuje kada se ubije). Promijenite metodu **`Arena.kill(Enemy)`** – ako je posljednji neprijatelj ubijen, igrač je dobio igru – zaustavite Greenfoot i napišite poruku na ekranu.

6. Nasljeđivanje

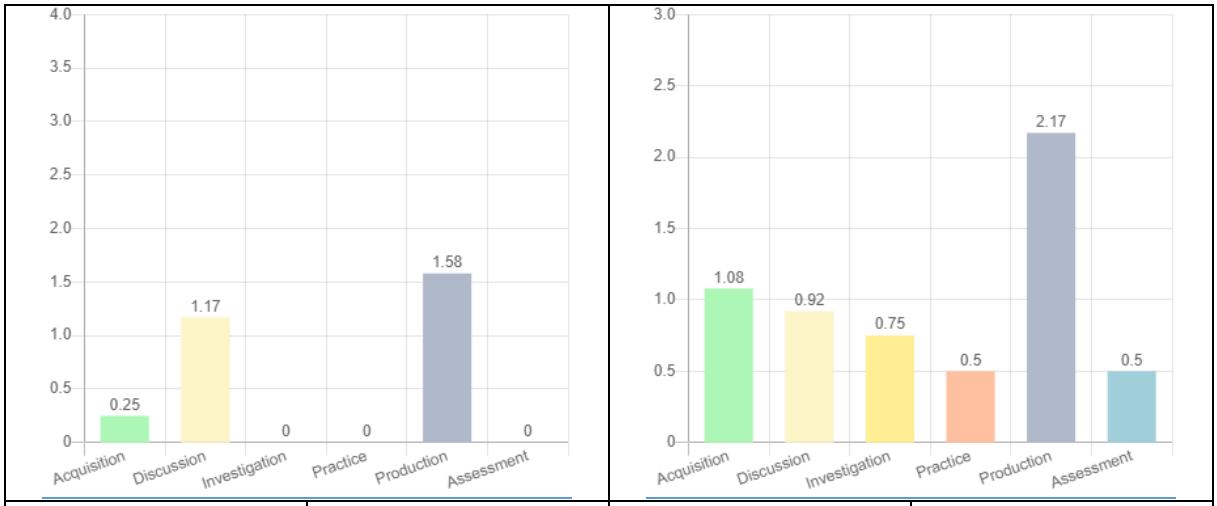
Ova tema uvodi varijabilnost u projekt putem naslijeđivanja. Uvodimo Liskov zamjenski princip kako bismo pokazali prednosti OOP-a. Snažno preporučujemo da učenici eksperimentiraju i osmisle

prilagođene podklase neprijatelja i arena. Budući da će ove imati zajedničko sučelje, bit će lako sve povezati. Slično kao u temi 4, moguće je stvoriti mnogo domaćih zadataka.

Kao što je već spomenuto, projekt se može smatrati završenim nakon prethodne teme. Stoga, ako je potrebno, nastavnik može prilagoditi ovu temu kako bi pokazao prednosti naslijeđivanja i s njime povezane univerzalnosti samo na ovdje predloženim hijerarhijama klase **`Arena`** ili **`Enemy`** To će dovesti do smanjenja vremena povezanog s ovom temom.

Tablica 10 sažima usporedbu radnih opterećenja teme Naslijeđivanje između projekata Bomberman i Tower defense. Prijedlog za eksperimentiranje projiciran je kroz više istraživanja, prakse i produkcije tipova TLA. Više teorije pokriveno je u ovoj temi s fokusom na Liskov zamjenski princip.

Tablica 10. Usporedba opterećenja teme Nasljeđivanje između projekata Bomberman i Tower defense



Bomberman	3h 0min	Tower defense	5h 55min

6.1. Identificirajte zajednička svojstva klasa Orb i Direction

*Instance klasa Orb i Direction ne djeluju tijekom životnog vijeka. Oni samo reagiraju na poruke. Možemo uvesti zajedničkog pretka koji će metodu **`act()`** držati praznom i učiniti podklase transparentnijima.*

6.2. Dodajte klasu PassiveActor kao pretka klasa Orb i Direction

Stvorite novu klasu **`PassiveActor`**. Promijenite kodove klasa **`Orb`** i **`Direction`** da budu potomci **`PassiveActor`**. Uklonite metodu **`act()`** iz klasa **`Orb`** i **`Direction`** – naslijeđena je od **`PassiveActor`**.

6.3. Kreirajte klasu PassiveActor abstract

Kada kreiramo metode klase **PassiveActor**, susrećemo metode koje se ne mogu implementirati u zajedničkoj klasi i stoga moramo ostaviti implementaciju podklasama. Kao primjer, razmotrimo klasu **Shape** s podklasama **Rectangle** i **Triangle**. Svaki oblik će imati metode **perimeter** i **content** implementirane, ali se ne mogu implementirati u zajedničkoj klasi. Ako označimo metodu klase kao **abstract**, u suštini kažemo da će je podklasa implementirati. Klasa koja sadrži apstraktnu metodu mora biti apstraktna. Stoga, dodajemo riječ **abstract** u zaglavlje klase.

6.4. Identificirajte zajednička svojstva klasa BulletiEnemy

*Instance klasa Bullet i Enemy djeluju slično tijekom života. Kreću se istim putem i nakon toga reagiraju na okolinu. Možemo uvesti zajedničkog pretka, koji će implementirati metodu **`act()`** da se kreće na isti način i da se podklase fokusiraju na svoju specifičnu svrhu.*

6.5. Dodajte apstraktnu klasu MovingActor kao pretka klasa Bullet i Enemy

Koristite sličan pristup kao u 6.2.

6.6. Identificirajte atribute klasa Bullet i Enemy potrebne za kretanje

Istražite metodu **`act()`** odgovarajućih klasa. Identificirajte atribute **`moveDelay`** i **`nextMoveCounter`**.. Primijetite, kod metode **`act()`** odgovorne za kretanje je isti.

6.7. Premjestite kod odgovoran za kretanje u klasu MovingActor

- Premjestite atribute identificirane u 6.6 iz podklasa **`Bullet`** i **`Enemy`** u **`MovingActor`** (uklonite ih iz podklasa).
- Dodajte parametarski konstruktor u klasu **`MovingActor`** za inicijalizaciju ovih atributa.
- Pozovite nadređeni konstruktor s odgovarajućim parametrima iz podklasa **`Bullet`** i **`Enemy`**.
- Premjestite kod odgovoran za kretanje u metodi a **`act()`** podklasa **`Bullet`** i **`Enemy`** u **`MovingActor`** (uklonite kod iz podklasa, zadržite ostatak metode).
- Pozovite nadređenu verziju metode **`act()`** kao prvu liniju metode **`act()`** u potklasama **`Bullet`** i **`Enemy`**.

6.8. Stvorite prilagođene neprijatelje

- Dodajte podklase klase **`Enemy`** koje će predstavljati različite neprijatelje (npr. **`Frog`** i **`Spider`**). Pazite da slike ne premašuju veličinu ćelije.
- Dodajte u klase konstruktor bez parametara, koji će pozvati nadređeni (**`Enemy`**'s) konstruktor s parametrima specifičnim za svaku vrstu neprijatelja.
- Uklonite metodu **`act()`** ((ili dodajte poziv **`super`**).

6.9. Pokreni prilagođene neprijatelje

Ažurirajte metodu **`Arena.spawn()`**. Napravite instancu **`Frog`** ili **`Spider`** i pohranite je u varijablu tipa **`Enemy`**. Upotrijebite bilo koju vrstu odluke da odlučite koju instancu treba kreirati (može biti nasumična, može biti precizno prebrojana, itd.). Pazite da se nijedan drugi kod u aplikaciji ne smije mijenjati

6.10. Razgovarajte o hijerarhiji arena

Raspravite i osmislite hijerarhiju **`Arena`** klasa. Potklase Arene bit će odgovorne za prilagođeni raspored – položaj **`Orb`** i **`Direction`** instanci, veličinu arene. Ovi zadaci će se obavljati u konstruktoru podklase. Što će biti potrebno za prijenos u parametre konstruktora roditeljske klase (**`Arena`**)? Imajte na umu da će sve ostalo (spawning, respawning i ubijanje neprijatelja) biti izvedeno u klasi **`Arena`**.

Trebali biste utvrditi potrebu za postavljanjem i pohranjivanjem položaja i rotacije pojava. To će se učiniti pomoću atributa i relevantnih parametara konstruktora. Štoviše, konstruktor također treba prihvatiti dimenzije površine.

6.11. Kreirajte univerzalnu Arenu

Uvedite atribute **`int spawnPositionX`** , **`int spawnPositionY`** i **`int spawnRotation`** i koristite ih u metodama **`spawn()`** i **`respawn(Enemy)`**.

Dodajte parametre u konstruktor klase Arena da ih inicijalizirate. Dodajte još dva parametra u konstruktor klase **`Arena`** – **`int width`** i **`int height`**. Proslijedite ove parametre nadređenom konstruktoru.

Primijetite da Greenfoot ne može automatski konstruirati Arenu jer su joj potrebni parametri za konstruktor. Neka napravite je **`abstract`**.

6.12. Kreirajte DemoArenu

Dodajte podklasu **DemoArena** od klase class **Arena**. Pozovite roditeljskog konstruktora klase **DemoArena** s parametrima koji će osigurati stvaranje arene istih dimenzija i stvaranje neprijatelja na isti način kao prije.

Premjestite kod odgovoran za raspored instanci klase **Direction**, **Orb** i **Tower** iz konstruktora **Arene** u konstruktor **DemoArena**.

Kreirajte instancu **DemoArena** – iz kontekstnog izbornika klase **class DemoArena** odaberite stavku nova **new DemoArena()**.

6.13. Stvorite prilagođene arene

Koristeći sličan pristup kao u 6.12 kreirajte druge inovativne podklase klase Arena. Možete podijeliti kod s ostalim učenicima u vašoj grupi.

7. Enkapsulacija

Posljednja tema je usmjerena na pravilno korištenje privatnih metoda i klasnih metoda i varijabli. Korištenje klasnih metoda i varijabli može se zamijeniti (ne-klasnim) atributima i metodama u klasi **Arena** (u kontekstu našeg projekta postoji samo jedna instanca klase **Arena**). To će omogućiti implementaciju zadataka iz ove teme koristeći već poznate koncepte.

Već smo koristili enkapsulaciju za projekt Bomberman. Ovdje ćemo koristiti statičke atribute i statičke metode. Do sada su svi atributi i metode bili povezani s instancom klase. Zamislimo situaciju u kojoj želimo brojati ispaljene metke u klasi **Bullet**. Broj ispaljenih metaka zajednički je svim instancama klase **Bullet** i ne ovisi o određenoj instanci. Takav atribut zovemo statički i budući da je zajednički za klasu, pristupa se preko imena klase, npr. **Bullet.count**. Na sličan način možemo stvoriti, na primjer, metodu koja vraća broj metaka. Opet, to će biti zajedničko svim instancama klase, stoga će biti statičko. Statički atributi i statičke metode imaju ključnu riječ **static** u svojoj deklaraciji. Imajte na umu da mogu postojati čak i ako ne postoji instanca klase. Statički atributi se inicijaliziraju u samoj definiciji.

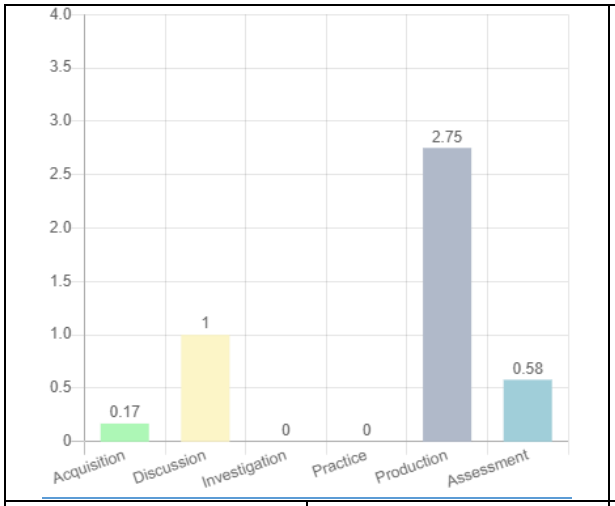
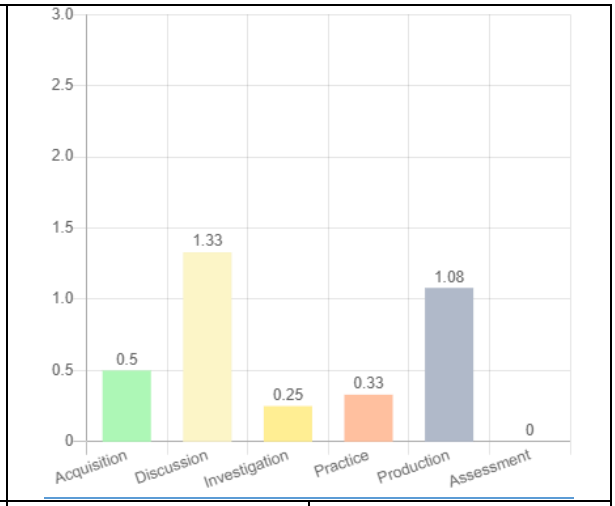
```
static int countOfBullets = 0;

static int countOfBullets(){

      ...
}
```

Tablica 11 sažima usporedbu opterećenja teme Enkapsulacija između projekata Bomberman i Tower Defense. Slično prethodnoj temi, produkcija je ravnomjernije raspodijeljena među ostalim vrstama TLA-ova. Veći broj akvizicijskih TLA-ova proizlazi iz uvođenja klasnih metoda i varijabli. Kao što je predloženo, postoji mogućnost izbjegavanja ovih koncepata, što će dovesti do smanjenja tih TLA-ova i do sličnog dizajna kao kod projekta Bomberman.

Tablica 11. Usporedba opterećenja teme Enkapsulacija između projekata Bomberman i Tower Defense

	Bomberman	4h 30min	Tower defense	3h 30min
			

7.1. Kreiranje klase ManualTower

Napravi klasu **`class ManualTowe`** kao potomak klase **`class Tower`**.. Pripremi slike za ovu klasu kada je pod kontrolom i kada nije. Dodaj dva konstruktora s istim potpisom kao konstruktori roditeljske klase i osiguraj pozivanje roditeljskog konstruktora. Neka metoda **`act()`** pozove roditeljsku verziju same sebe.

Dodaj instance ove klase u raspored odabrane **`Arena`**.

7.2. Promjena kontorle ručno upravljanog tornja

Dodaj atribut **`boolean isManuallyControlled`** i inicijaliziraj ga na **`false`**. Napravi metodu **`void changeControl(boolean)`** koja mijenja atribut i mijenja sliku u skladu s tim.

7.3. Pozovi promjenu ručne

Ručnim putem pozovi promjenu ručne kontrole odabrane instance **`class ManualTower`**. Promatraj promjene u unutarnjem stanju slično kao u 1.6.

7.4. Obrada korisničke kontrole

Napravi privatnu metodu **`void processUserControl()`**. Prvo detektiraj je li kliknuto na ovu instancu. Ako jest, promijeni na ručnu kontrolu. Zatim implementiraj samu ručnu kontrolu. Testiraj je li instanca u ručnom načinu rada i ako jest, pribavi objekt **`MouseInfo`**. Ako je objekt pribavljen, okreni instancu **ManualTower** prema poziciji kursora miša.

Pozovi pripremljenu metodu **`procesUserControl()`** iz metode **act()** prije nego što se izvršenje proslijedi roditelju (**super.act()**). Provjeri u metodi je li kliknuto na ovu instancu. Ako jest, pozovi metodu **`changeControl`** s odgovarajućom vrijednošću.

Testiraj svoje rješenje izvršavanjem 7.3. ponovno.

7.5. Identificiranje problema s korisničkom kontrolom i prijedlog rješenja

Identificiraj što je problematično s korisničkom kontrolom. Kako se ti problemi mogu riješiti?

Trenutno nije moguće poništiti odabir tornja. Trebao bi postojati dokaz o trenutno kontroliranoj instanci, koja će biti deaktivirana kada se odabere neka druga instanca. Dodaj dokaz o ručno kontroliranom tornju.

7.6. Dodavanje dokaza o ručno kontroliranom tornju

Dodaj atribut u klasu **ManualTower** tipa **ManualTower** koji će predstavljati referencu na instancu ručno kontroliranog objekta i inicijalizirajte ga na **null**. Ovaj atribut mora biti zajednički za sve instance klase **ManualTower**, stoga ga treba definirati s ključnom riječi **static**. Provjeri unutarnje stanje klase (iz kontekstnog izbornika klase **ManualTower** odaberite stavku **Inspect**). Što je dodano?

7.7. Promjena ručno kontroloiranog tornja s centralnog mjesta

Dodaj metodu **`changeControlledInstance`** kao klasnu metodu klase **`class ManualTower`** (zbog toga definiranu primjenom ključne riječi **static**) za promjenu ručno kontroliranog tornja. Parametar metode treba biti referenca na instancu **`ManualTower`** koja će biti ručno kontrolirana.

Ako se parametar metode razlikuje od ručno kontrolirane instance (u statičkom atributu), koristite metodu **changeControll** s ispravnim parametrima. Prvo, koristite **changeControll** za postavljanje izvorne ručno kontrolirane instance na **false**. Zatim promijenite statički atribut klase **ManualTower** ručno kontrolirane instance na parametar nove metode (tj. novu ručno kontroliranu kulu). Na kraju, koristite metodu **changeControll** za postavljanje nove ručno kontrolirane instance na **true**. Ne zaboravite obraditi moguće **null** reference. To je važno jer može doći do situacije u kojoj ne postoji ručno kontrolirana kula. Ako se prethodno i novo proslijeđene instance razlikuju, pošaljite poruku **changeControll** s odgovarajućim parametrima prethodnoj, kao i novoj pohranjena referenca na instancu klase **ManualTower**. Ne zaboravite obraditi **null** reference i pohraniti novu referencu!

Testiraj svoje rješenje. Iz kontekstnog izbornika klase **`class Tower`** odaberi stavku s novom stvorenom metodom. Za unos parametra možeš koristiti isti princip kao u **Error! Reference source not found.**.

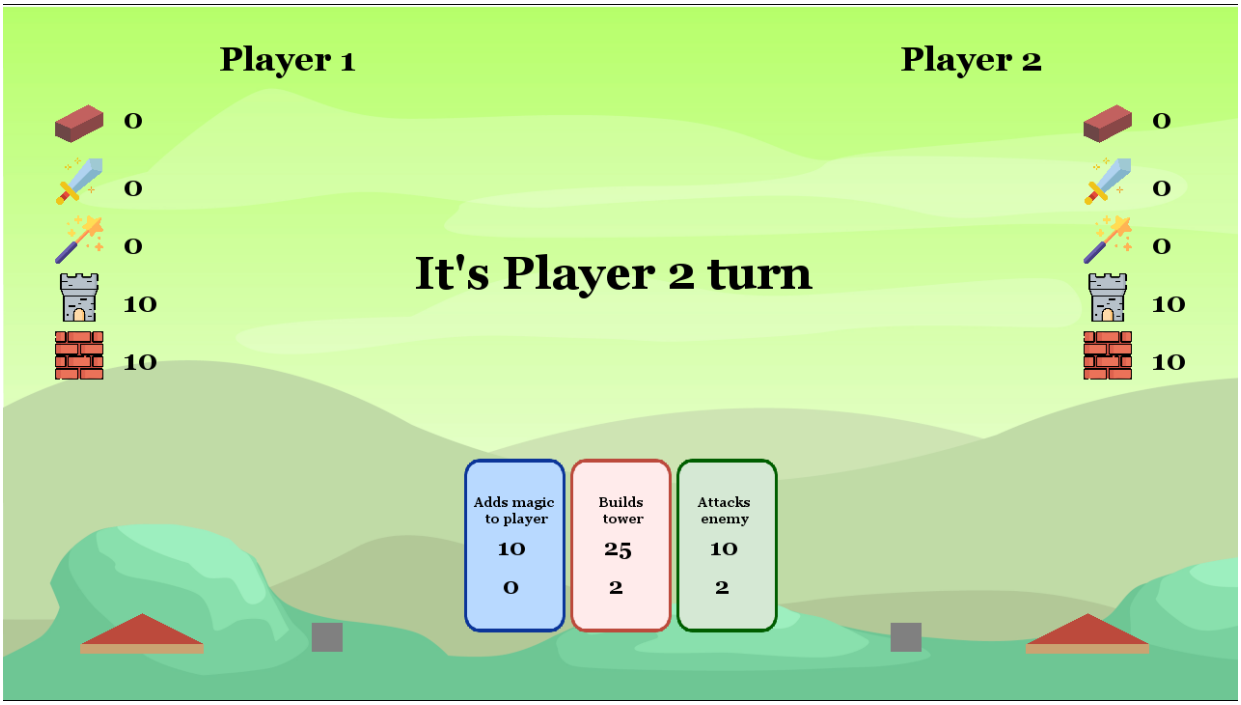
*Trebali biste primijetiti da novostvorena klasna metoda nije dosljedno pozvana. Instance klase **`ManualTower`** zaobilaze dokaz kada obrađuju unos, što uzrokuje probleme.*

7.8. Pozovi promjene ručno kontroliranog tornja

Pozovi **`ManualTower.changeControlledInstance(ManualTower)`** s odgovarajućih mjesta. Na kraju, metodu **`ManualTower.changeControl(boolean)`** učini privatnom. Promatraj promjene sučelja instance **`class ManualTower`** slično kao u to **Error! Reference source not found.**.

3.3. Projekt Ants

"Ants" je kartaška igra za dva igrača. Svaki igrač ima svoj vlastiti toranj, zid i resurse poput cigli, mačeva i magije. Jedan potez u igri sastoji se od akcije u kojoj igraču igra ponudi 3 slučajno odabrane karte, od kojih on odabire jednu. Postoje tri vrste karata – građevinske karte, borbeni karti i magične karte. Građevinske karte se mogu koristiti za pojačavanje vlastitog tornja ili zida, borbeni karti za napad na protivničkog igrača, a magične karte za povećanje broja vlastitih resursa ili krađu protivnikovih.



Izvorni kod dostupan na:
https://gitlab.kicon.fri.uniza.sk/oop4fun/project-ants

Dizajn učenja dostupan na:
http://learning-design.eu/en/preview/67aa1d089763d07f29809d42/details

3.3.1. Teme

Projekt Ants is je podijeljen u šest tema:

Pokrivene teme laganog OOP-a (objektno orijentirano programiranje) su:

- klase, objekti, instanca
- metode, proslijeđivanje argumenata metoda
- konstruktori

- atributi
- statičke varijable i metode
- enkapsulacija

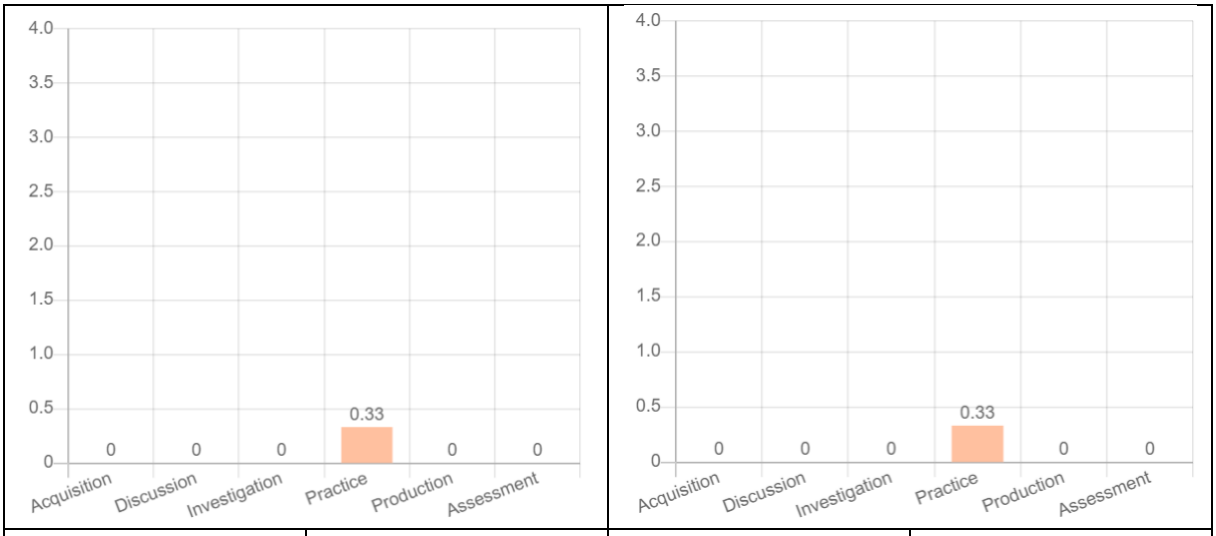
1. Okruženje Greenfoot, definicija klase, osnovni rad s klasama

Tema je posvećena stvaranju projekta. Učenici će biti sposobni stvoriti novi projekt u Greenfoot okruženju, kreirati klasu (kao podklasu **`Actor-a`**), odabrati pozadinsku sliku, kreirati instancu te klase i poslati joj poruku.

Kreirajte novi projekt. Dodijelite mu odgovarajuće ime (npr. **`Ants`**) i spremite ga na odgovarajuće mjesto.

Tablica 12 sažima usporedbu opterećenja prve teme između projekata Bomberman i Ants. Ukupno opterećenje oba projekta je isto u ovom dijelu.

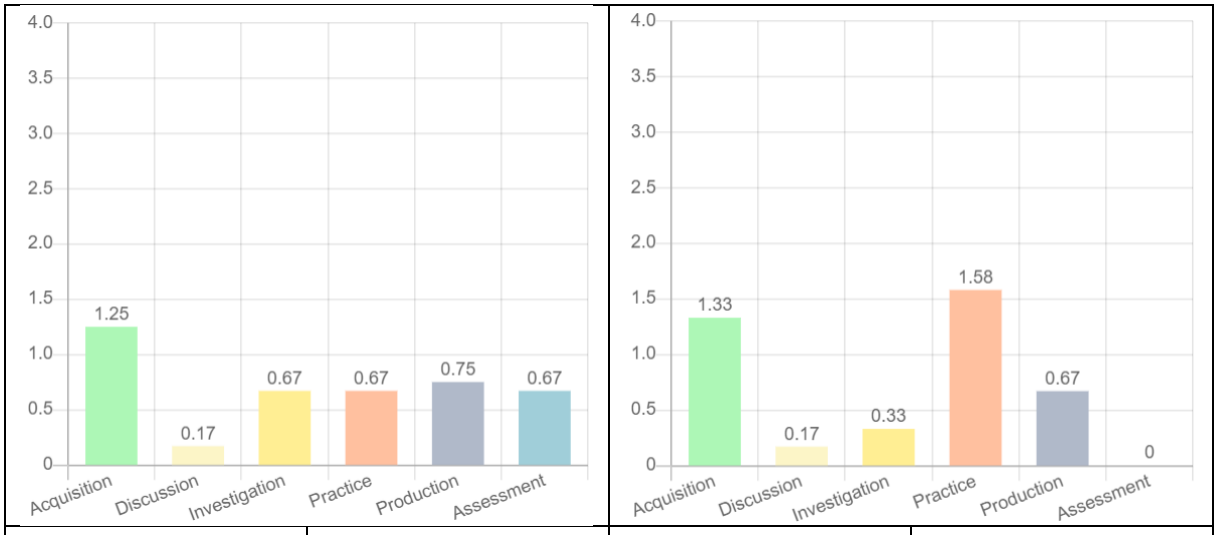
Tablica 12. Usporedba radnih opterećenja teme 1 između projekata Bomberman i Ants



Bomberman	0h 20min	Ants	0h 20min

Tablica 13 sažima usporedbu opterećenja teme Definicija razreda i osnovni rad s razredima između projekata Bomberman i Ants. Ukupno opterećenje oba projekta je slično. Glavna razlika je u praksi i ocjenjivanju. Međutim, kako će biti navedeno u sljedećem odjeljku, neke od praksi mogu se također koristiti kao zadaci za ocjenjivanje, što može dodatno izbalansirati situaciju.

Tablica 13. Usporedba radnog opterećenja za temu definirajne klasa i osnovni rad s klasama između projekata Bomberman i Ants



Bomberman	4h 10min	Ants	4h 05min

1.1. Uvod u Greenfoot

Stvorite novi projekt u Greenfootu i upoznajte studente s osnovnim elementima, korisničkim sučeljem i sl. Početno stanje repozitorija također sadrži resurse koje možete koristiti u projektu.

1.2. Kreiranje klase class Wall

Kreirajte razred **`Wall`** kao podklasu **`Actor-a`**. Upoznajte učenike s konceptima kao što su klase, hijerarhija klasa, instance itd.

1.3. Kreiranje klase class Tower

Slično kao u prethodnom zadatku, kreirajte klasu **Tower**. Možete ga ostaviti učenicima kao zaseban zadatak.

1.4. Defininiranje atributa klase/polja

Upoznajte učenike s pojmovima polje/atribut, primitivni tipovi podataka itd. Pokušajte identificirati polja u našim razredima (posebno usmjerite učenike na visinu- **`height`**) i definirajte ga u kiasi **`Wall`**.

1.5. Dodijeljivanje vrijednosti polju/atributu

Razgovarajte o vrijednostima polja i dodjelama te dodjelite **`height`** visini zida vrijednost 10.

1.6. Definiranje i dodijeljivanje vrijednosti atributu/polju za klasu Tower

Kao individualni zadatak, ostavite učenicima da ponove isto za klasu **`Tower`**.

1.7. Konstruktori klasa

Razgovarajte o instanciranju klase i konstruktorima. Premještanje dodjele vrijednosti u konstruktor znači dodjeljivanje vrijednosti tijekom stvaranja objekta.

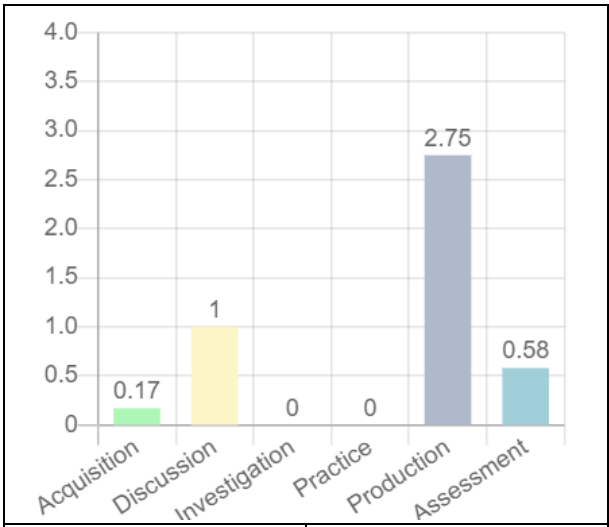
2. Enkapsulacija, kompozicija, metode

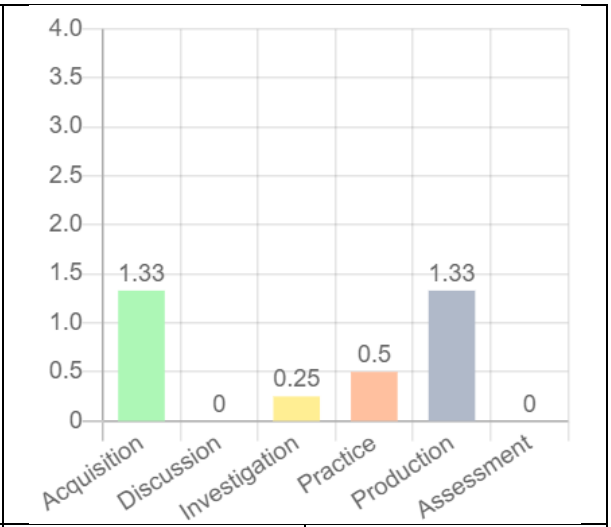
Ovo poglavlje pruža osnovne principe objektno orijentiranog programiranja (OOP), posebno pojmove kao što su inkapsulacija, kompozicija i metode. Učenici će naučiti kako i zašto polja/atributi trebaju biti

inkapsulirani, a ne javno dostupni, kako se objekti komponiraju s drugim objektima te kako kreirati metode i pozivati ih. Učenici će u ovom poglavlju stvoriti objekt tipa **`Player`** te mu dodijeliti polja tipa objekta – **`Wall`** i **`Tower`**.

Tablica 14 prikazuje razlike između dva slična tematska područja - Inkapsulacija u projektu Bomberman i Inkapsulacija, kompozicija i metode u projektu Ants. Kao što se može vidjeti, projekt Ants više se fokusira na stjecanje znanja, dok se projekt Bomberman više usredotočuje na proizvodnju i raspravu. To je posljedica činjenice da tema u projektu Ants obuhvaća više od same inkapsulacije, stoga je potrebno i više stjecanja znanja. Ukupan radni teret je sat vremena kraći u projektu Ants u usporedbi s projektom Bomberman jer su ovi koncepti objašnjeni manje dubinski, ali će se znanje u idućim sekcijama dodatno utvrditi.

Tablica 14. Usporedba radnog opterećenja teme Enkapsulacija, kompozija i metode između projekata Ants i slične teme Enkapsulacija u projektu Bomberman





Bomberman	4h 30min	Ants	3h 25min

2.1. Definiranje metoda

Razgovarajte o enkapsulaciji, objasnite učemnicima zašto nije dobra praksa na primjer napraviti visinu zida **`height`** kao javno svojstvo i radije je inkapsulirati u getteru. Zatim stvorite metodu **`getHeight()`** in **`Wall`**.

2.2. Definiranje metoda s parametrima

Objasnite parametre metode i definirajte metodu i **`increaseHeigh`** u klasi Wall koja će povećati visinu zida **`height`** za određeni broj.

2.3. Ponavljanje za kalsu Tower

Kao individualni zadatak možete zadati učenicima da ponove isto i za klasu **`Tower`**.

2.4. Kompozicija objekta

Objasnite učenicima što je kompozicija i zašto je potrebno imati tipove objekata kao polja. Pokušajte identificirati takve vrste za svakog igrača u ovoj igri. Zatim stvorite objekt **`Player`** i dodijelite mu polja/atribute **`Wall`** i **`Tower`**

2.5. *Kreiranje istance klase*

Objasnite konstruktore i kreirajte **`Wall`** i **`Tower`** instance u **`Player`** konstruktoru.

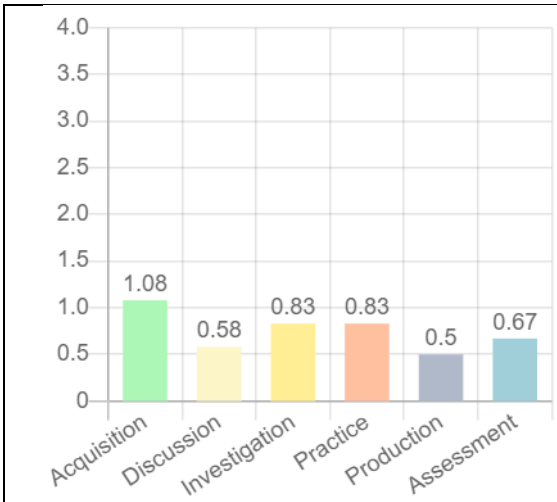
2.6. *Pozivanje metode instance*

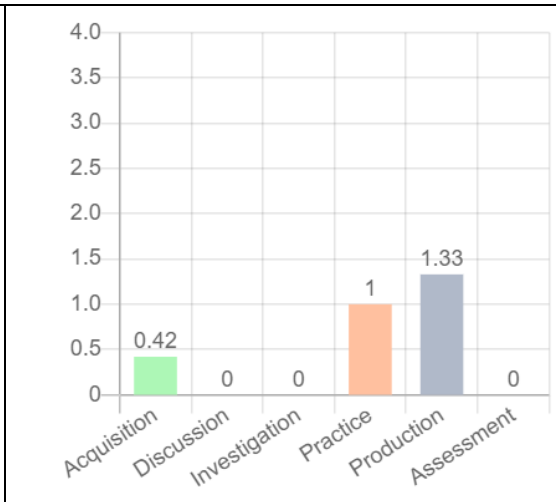
Objasnite učenicima kako mogu pozvati metode stvorenih instanci objekata. Zatim pokušajte ih inkapsulirati tako da ih se može pozivati izvana kroz instancu Player. Stvorite metode **`getWallHeight`**, **`getTowerHeight`**, i **`increaseWallHeight`**, **`increaseTowerHeight u`** objektu **`Player`** object.

3. Konstruktori, složeniji pozivi metoda (rad s grafičkim elemntima u Greenfootu)

Ovo poglavlje usmjereno je na pozive metoda koristeći pozivanje Greenfoot objekata za crtanje naših instanci. Učenici će crtati instance zida, tornja i igrača.Tablica 15 prikazuje usporedbu ovog tematskog područja u projektu Ants i najsvrsnije sličnog tematskog područja u projektu Bomberman. Molimo, imajte na umu da su ovi tematski sadržaji malo različiti, jer naš prikazuje osnovni rad s konstruktorima i uvodi grafiku u Greenfoot, dok je u projektu Bomberman fokus više na algoritmima. Stoga se mogu primijetiti i razlike zbog ovih razlika.

Tablica 15. Usporedba padnog opterećenja teme Konstruktori, složenijih poziva metoda (rad s grafičkim elementima u Greenfootu) između projekata Ants i slične teme u projektku Bomberman koja je pokrivena u temi Algoritmi





Bomberman	4h 30min	Ants	2h 45min

3.1. *Crtanje objekata u Greenfootu – Zid*

Upoznajte učenike s konstantama u Javi – definirajte **wallSizeX** i **wallSizeY** kao **static final** (ključna riječ **final** osigurava da vrijednost ne može biti promijenjena, tj. da je konstantna) atribute (konstantne) s vrijednostima **32** i **3**. Konstantama se pristupa kao statičkim varijablama preko imena klase, npr. **Wall.wallSizeX**. Zatim implementirajte funkciju **redraw** u klasi **Wall**, gdje se stvara nova slika sa zadanim veličinama i ispunjava kao pravokutnik.

3.2. *Crtanje objekata u Greenfootu – Kula*

Isto se ponavlja za **`Tower`** međutim, to je složenije jer **`Tower`** također uključuje krov koji je implementiran kao poligon. Poligon zahtijeva niz točaka. Ako želite, možete učenicima kratko objasniti, iako nizovi nisu dio ovog odjeljka. Razgovarajte s učenicima o korištenju termina **array** I **list**.

3.3. *Definiranje drugih osobina Playera (Igrača)*

Pokušajte identificirati druge osobine igrača - **`Player`** - posebno broj cigli, mačeva i magija te ime igrača- **`Player`**. Zatim implementirajte ove osobine u klasi **`Player`** i inicijalizirajte ih u konstruktoru.

3.4. *Crtanje Playera (igrača)*

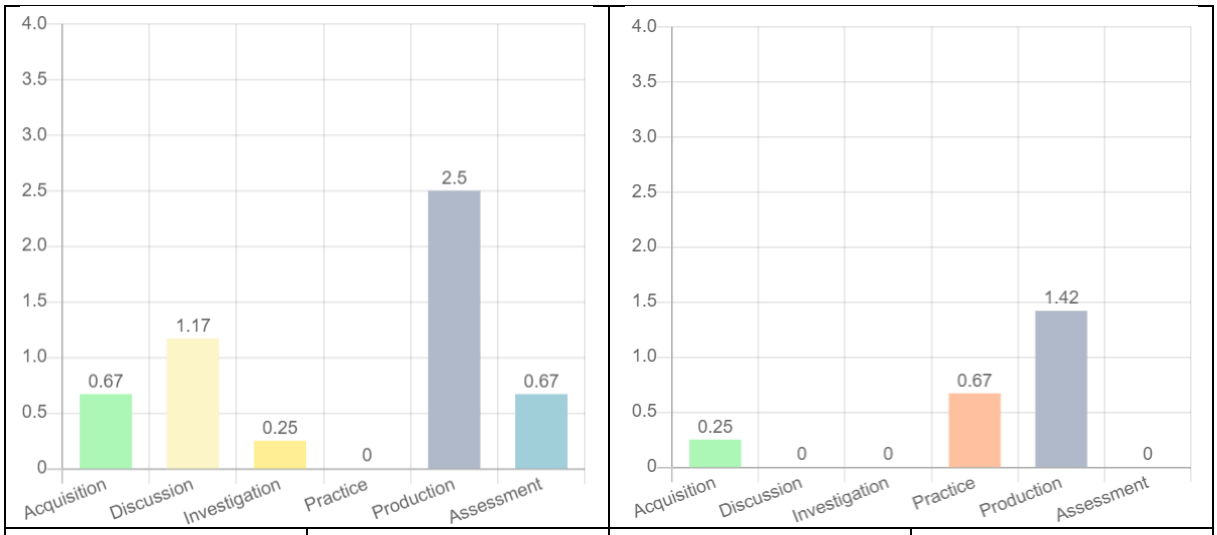
Zadnji zadatak u ovom dijelu je crtanje igrača- **`Player`**. Trebate nacrtati ikone svakog resursa, broj tog resursa te također ime igrača.

4. Grananje, uvjetno izvršavanje

Ovo poglavlje fokusira se na grananje programa i uvjetno izvršavanje dijelova naše igre. Postoje nekoliko slučajeva gdje je to potrebno u ovoj igri, kao što je crtanje prvog igrača na lijevom dijelu ekrana i drugog na desnom itd.

Tablica 16 prikazuje razlike između sličnih tema u projektima Bomberman i Ants. Kao što se može vidjeti, projekt Bomberman daje veći naglasak na raspravu nego projekt Ants. Također, ukupno radno opterećenje u satima je približno polovica onog u projektu Bomberman. To je zbog činjenice da je grananje podijeljeno u više dijelova u ovom projektu - ovo poglavlje služi kao uvod i osnovna upotreba grananja.

Tablica 16. Usporedba radnog opterećenja za temu Grananje i uvjetno izvršenje između projekata Bomberman i Ants



Bomberman	5h 15min	Ants	2h 20min

4.1. *Kreitranje objekta Game (Igre)*

Prije nego što počnemo implementirati grananje u naš kod, trebamo stvoriti objekt **`Game`** koji će držati oba igrača i u budućnosti će upravljati logikom igre, promjenom poteza, izvršavanjem karata itd. Za sada ćemo tamo staviti svojstva za dva igrača i kreirati njihove instance u konstruktoru klase **Game**.

4.2. *Grananje, uvjetno izvršavanje koda – igrači su prikazani na odgovarajućim stranama igračkog plana*

Sada trebamo uvesti novi atribut za našeg igrača- **`Player`** – informaciju treba li igrač biti prikazan na lijevoj ili desnoj strani ekrana. Tu informaciju ćemo koristiti za postavljanje odgovarajućih svojstava igrača – položaj za **`Wall`**, **`Tower`**, "hud" i **`name`**. Ovo odstupanje treba zatim dodati u funkciju za ponovno crtanje- **`redraw.`**

4.3. Dodavanje instanci u svijet

Sljedeći zadatak stvara početni prikaz instanci klase **Player** za igru i inicijalizira igru u svijetu.

4.4. Dodavanje instanci u svijet 2

Da bi se igrač - **`Player`** pravilno nacrtao u igri- **`Game`**, moramo dodati Zid- **`Wall`** i Kulu- **`Tower`** u svijet te pozvati funkciju za ponovno crtanje **`redraw`** u metodi **`act`**. Ovdje možete objasniti izvršenje petlje igre (**`act`** method).

4.5. Izvršavanje koda samo jednom

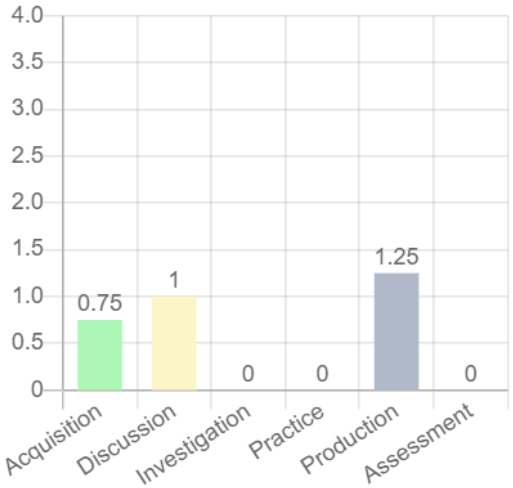
Kada pokušate pokrenuti igru **`Game`** nakon posljednjeg zadatka, možete naići na problem – objekti se dodaju u svijet više puta u sekundi. Možete dati učenicima zadatke za rješavanje ovog problema ili ga riješiti s njima. Jedno od rješenja je uvođenje novog boolean atributa koji će pohranjivati informaciju je li se početni objekt već dodao u svijet ili nije, i nakon prvog izvršenja metode act, postavlja se ova svojstva na true.

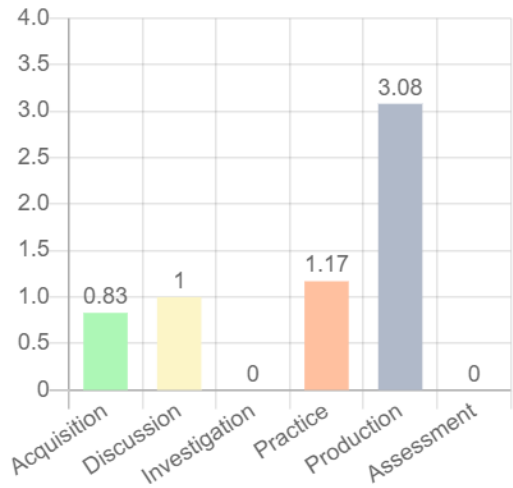
5. Algoritmi, enumeracije i polja

Ovaj odjeljak raspravlja o sljedećim konceptima, kao što su algoritam, enumeracije, polja i petlje preko polja elemenata. U ovom odjeljku, učenici će implementirati karte koje će se koristiti u igri.

Tablica 17 sadrži usporedbu sličnih tema u projektima Ants i Bomberman. Kako se može vidjeti, projekt Ants više se fokusira na proizvodnju i praksu, te malo više na stjecanje vještina. Također, ukupno radno opterećenje je otprilike dva puta veće nego u projektu Bomberman. To je uzrokovano činjenicom da ovaj projekt uključuje više od rada s listama, već uvodi i koncepte kao što su enumeracije itd.

Tablica 17. Usporedba radnog opterećenja teme Algoritmi, enumeracije i polja u prrojektu Ants i slinoj temi u projektu Bomberman – Liste





Bomberman	3h 00min	Ants	6h 05min

5.1. Implementacija klase kartica

Prvo možete raspravljati s učenicima o tome kako bi konačna igra trebala funkcionirati i dizajnirati objekt Kartica- **`Card`**. Trebali bismo doći do rješenja koje će sadržavati informacije o vrsti kartice, njezinim zahtjevima, učinku i nekom opisu. Zatim možemo kreirati takav objekt kao podklasu **`Actor`**.

5.2. Enumeracija

U prethodnom zadatku, stvorili ste atribut tipa karte kao polje u objektu **Card**. Razgovarajte sa studentima o tome koji tip bi to polje trebali biti – **String**, **int**, itd., i možete im predstaviti **enum** (enumeraciju). **Enum** je tip podataka s konačnim skupom imenovanih vrijednosti (npr. za dane u tjednu to su vrijednosti: **Monday**, **Tuesday**, **Wednesday**, **Thursday**, **Friday**, **Saturday** i **Sunday**). Kreirajte **CardType enum** s njima i navedite njegove pojedinačne vrijednosti.

5.3. Grananje – switch

Sada bismo trebali implementirati crtanje kartice- **`Card`**. To se temelji na vrsti kartice, kako bismo vizualno podijelili kartice. Možete pokazati učenicima kako bi se ovo moglo učiniti koristeći **`if`** i usporediti s prekidačem (**`switch`**). Postoje tri vrste kartica – građevinske kartice, napadačke kartice i magične kartice. Na temelju ove kategorije, odabire se pozadina. U slučaju da to radimo koristeći **`if`** izjavu, kod bi trebao izgledati ovako:

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

Ovo se može zamijeniti prekidačem **`switch`** statement.. Java prekidač radi na način da će se izvršiti određena grana, ali neće zaustaviti izvršavanje ostalih grana osim ako ne pozovete **`break`** naredbu . U našem slučaju, ovo nam omogućuje da spojimo više grana zajedno i napišemo dodjelu vrijednosti samo za posljednju vrstu kartice te kategorije. Stoga, ovaj kod:

```
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
      ...
}
```

može se napisati i na način koji koristimo u ovom zadatku:

```
switch (type) {
```

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

5.4. Polje

Objekt igre **Game** trebao bi sadržavati kartice- **Cards**. To se može postići uvođenjem tri polja tipa Card (možete proširiti i ruku – tj. broj kartica koje **Cards Game** nudi igraču - **Player** u jednom potezu – na više njih). Možete im objasniti da bi bilo nemoguće pohraniti još više kartica kad odlučimo proširiti ruku još više te im možete objasniti koncept polja (arrays). Također biste trebali stvoriti primjerak polja

5.5. Pojednostavljanje instanciranja kartica – CardFactory dFactory

Prilikom stvaranja novih kartica **Cards** potrebno je pružiti puno informacija. Možete pokušati pronaći rješenje za ovaj problem. Jedno od rješenja je stvaranje **CardFactory** (tvornica kartica) koji će držati primjerke - svake kartice i implementirati metodu kloniranja **clone** na **Card** objektu. Na taj način, nove **Cards** mogu se stvarati korištenjem ovog **CardFactory** i kloniranjem postojećih.

5.6. Slučajno instanciranje slučajne kartice

Nakon implementacije **CardFactory**, također treba implementirati metodu kloniranja, kako je već raspravljeno. **CardFactory** bi također trebao moći pružiti primjerke slučajne **Card** - kartice. Možete objasniti generator slučajnih brojeva i implementirati metodu koja će klonirati slučajnu karticu te slučajnu baznu karticu (bazne kartice su kartice bez troška - to je implementirano kako bismo osigurali da igrač- **Player** uvijek može igrati barem jednu od ponuđenih **Cards**.

5.7. Petlje u polju

Sada moramo stvoriti instancu **CardFactory** u igri i implementirati generiranje **Cards** te njihovo uklanjanje (brisanje iz svijeta) - prilikom objašnjavanja ovoga, možete uvesti neku vrstu petlje.

5.8. Crtanje Game (Igre)

Zadnji zadatak u ovoj sekciji je implementirati crtanje cijele **Game**. Prilikom toga, trebali bismo pripremiti naše **Cards** koristeći prethodno stvorenu metodu te također uvesti informaciju je li prvi igrač aktivan. Crtanje **Game** (igre) trebalo bi uključivati crtanje svih **Cards** (kartica) i prikaz informacija o tome koji igrač je trenutno na potezu.

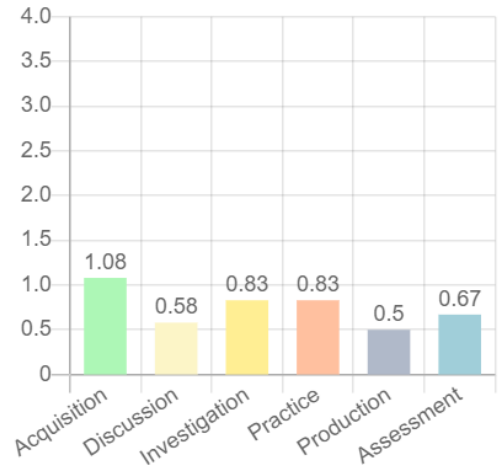
6. Upravljanje korisničkim unosom, logika igre

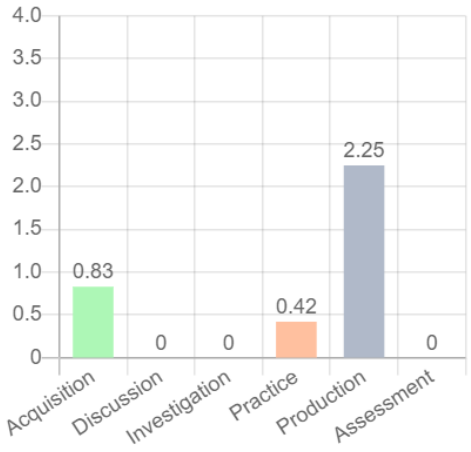
Ova sekcija fokusira na upravljanje korisničkim unosom – kako dobiti, primjerice, ime igrača od korisnika, kako obraditi klikanje na kartice te kako završiti logiku igre. U ovoj sekciji također se uvode neki napredni koncepti (singleton).

Tablica 18 prikazuje usporedbu između ove teme i najsličnije teme u projektu Bomberman – Algoritam. Imajte na umu da budući da je ovo posljednja tema ovog projekta, mnogi koncepti uvedeni u projektu

Bomberman već su poznati učenicima u ovom trenutku ovog projekta. Stoga, fokus na istraživanje i raspravu je znatno manji nego u projektu Bomberman, a više se naglašava produkcija.

Tablica 18. Usporedba radnog opterećenja teme Upravljanje korisničkim unosom, logika igre u projektu Ants i slične teme u projektu Bomberman- Algoritmi Konstruktivno usklađivanje projekta Bomberman





Bomberman	4h 10min	Ants	3h 30min

6.1. Unos imena od strane korisnika

U ovom zadatku trebate objasniti dijaloške prozore (metoda ask u Greenfootu) učenicima i postaviti imena igrača - **`Player`** prema tim unosima.

6.2. Statička instanca klase – Game kao singleton

Objekti igre **`Game`** bi trebali biti konstruirani točno jednom - ovaj problem možete raspraviti sa učenicima te možete ponuditi rješenje u obliku singletona - statične instance objekta **`Game`** i **`private`** konstruktora. Singleton se koristi u slučaju kada želimo samo jednu instancu neke klase u cijeloj aplikaciji. Poznat primjer singletona je npr. Paket u operacijsom sustavu. Ovo je potrebno jer kada igrač **`Player`** koristi kartu, treba postojati referenca na **`Game`** kako bi se klik na nju mogao pravilno obraditi, što će se raspraviti u sljedećem dijelu. Drugi način implementacije (bez singletona) je pružiti instancu **`Game`** u konstruktoru Card i **`CardFactory`**.

6.3. Obrada unosa klikom na kartu

Klik mišem se obrađuje pozivom metode **`Greenfoot.mouseClicked`** u metodi act. Kada se klikne, trebate pozvati metodu **`useCard`** klase **`Game`** i poslati referencu na sebe (**`this`**).

6.4. Implementiranje gettera za Card i Player

Prije nego što nastavimo s implementacijom ostalih logičkih dijelova igre, trebamo dodati još neke gettere i settere za **`Player`** i **`Cards`** Implementacija ovih metoda ne bi trebala predstavljati problem učenicima, budući da je već bila obavljena ranije.

6.5. IImplementiranje podržavajuće metode za Player i ispravak u World-u

Budući da je **Game** sada singleton, moramo ga dodati u svijet (World) u klasi **MyWorld**. Sljedeći zadatak je implementacija podržavajuće metode za igrača koja će se koristiti za primanje određenog iznosa štete. **Player** bi trebao smanjiti **Wall** ili **Tower** prema primljenoj šteti.

6.6. Implementiranje logike igre

Konačno, moramo implementirati logiku igre - metodu **turn** koja će se sastojati od sljedećih koraka:

1. Provjeriti pobjednika - to znači provjeriti je li visina tornja jednog od igrača dosegnula 100 ili pala ispod vrijednosti 0. Ako je jedan od ovih uvjeta ispunjen, prikazat ćemo zaslon pobjede i izaći iz petlje igre - pozvati **return** naredbu.
2. Postaviti aktivnog igrača na drugog - samo invertirati vrijednost atributa **isPlayer1Active**.
3. Pripremiti karte za sljedećeg igrača - budući da smo već implementirali metodu **prepareCards**, sve što moramo učiniti u ovom koraku je pozvati tu metodu.
4. Obraditi potez igrača - posebno klikom na karte. Kako sama karta **Card** može slušati klikove na nju, sve što moramo učiniti je pozvati metodu **draw** igre, koja crta pripremljene karte.
5. Ponovno iscrtati igrače - ovo se radi korištenjem metode **redraw** za svakog igrača

Zatim moramo implementirati metodu **useCard** u klasi **Game** koristeći **switch**. Ovaj **switch** treba sadržavati granu za svaki tip karte i obraditi njeno izvršenje. Postoji 7 vrsta karata: **BuildTower**, **BuildWall**, **IncreaseBricks**, **IncreaseSwords**, **Attack**, **IncreaseMagic** i **StealBricks**. Pogledajmo prvo vrstu karte - **BuildTower**. U ovom slučaju moramo provjeriti može li igrač odigrati ovu kartu (tj. broj njegovih cigli je veći ili jednak zahtjevima karte), povećati visinu njegovog tornja **Tower height** i smanjiti broj njegovih cigli **bricksNumber** za zahtjeve karte. Dakle, kod može izgledati ovako:

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

Drugi tipovi karata su slični:

- **BuildWall** – moramo provjeriti igračev broj cigli **bricksNumber** i povećati visinu zida **Wall height**.
- **IncreaseBricks** – ne moramo ništa provjeravati i povećavamo **bricksNumber**
- **IncreaseSwords** – ne moramo ništa provjeravati i povećavamo broj mačeva **swordsNumber**.
- **Attack** – moramo provjeriti igračev broj mačeva **swordsNumber** i pozvati metodu **receiveDamag** neaktivnog igrača.

- **`IncreaseMagic`** – ne moramo ništa provjeravati i povećavamo **`magicNumber`**
- **`StealBricks`** – moramo provjeriti **`magicNumber`**, smanjiti broj cigli b**`ricksNumber`** neaktivnog igrača i povećati broj cigli **`bricksNumber`** aktivnog igrača.

Moguće je kreirati i druge vrste karata - to je na mašti studenata. Ovo su neke osnovne vrste.

Konačno, nakon što odigramo kartu, moramo pozvati metodu turn kako bismo osigurali logiku igre.

Kao posljednji korak, trebamo implementirati zaslon pobjede. Ovo je slično ostalim crtežima u projektu, pa je na vama hoćete li ga kreirati zajedno sa učenicima ili ga prepustiti njima.

Za kraj, trebaju se napraviti i neke ispravke u **`Player i Tower`** radi čišćeg koda.

4. Literatura

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

[2] „GIT, SVN, mercurial – Google Trends,“ 2 10 2023. [Online]. Dostupno: 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].Dostupno: https://github.com. [Cit. 1 10 2023].

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

5. Prilozi

5.1. Izvoz dizajna učenja za projekt Bomberman

Pogledaj datoteku LD_Bomberman.pdf

5.2. Izvoz dizajna učenja za projekt Tower defense

Pogledaj datoteku LD_Tower_defense.pdf

5.3. Izvoz dizajna učenja za projekt Ants

Pogledaj datoteku LD_Ants.pdf