Academic Year 2017-2018

G52SWM

Understanding and improving other people's software

4302178

Wang Jinhao

(Total word: 992)

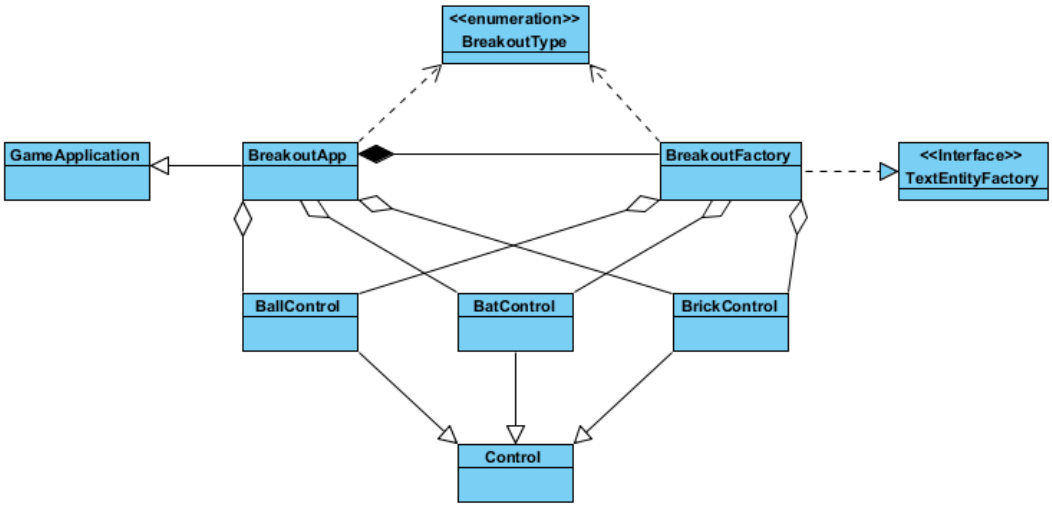
Introduction

At present, with the development of programming tools and software constructions, codes are becoming larger and larger in many software applications. It is not a surprise today that a simple mobile game can contain thousands of rows of code in its small installation package.

However, developing a software doesn't mean just making the application run without exceptions. Software maintenance and extension are also important aspects of software development. Therefore, software engineers are paying more and more attention to making good code. Since maintenance and extension are usually done by different development team, producing a good code has become an essential part of software programming.

Making bad codes good is also a basic skill which all programmers should achieve. Before improving one's codes, the first thing needs to do is to understand the codes. In this situation, a UML Class diagram may be helpful for us to better understand other's codes.

0. UML Class Diagram



(1) BallControl, BatControl, BrickControl inherit from Control

```
public class BallControl extends Control
public class BatControl extends Control
public class BrickControl extends Control
```

(2) BreakoutApp inherits from GameApplication

```
public class BreakoutApp extends GameApplication
```

(3) BreakoutFactory inherits from TextEntityFactory, which is an interface

```
public class BreakoutFactory implements TextEntityFactory
```

(4) BallControl, BatControl, BrickControl are parts of BreakoutApp (Aggregation)

```
private BatControl getBatControl() {
    return getGameWorld().getSingleton(BreakoutType.BAT).get().getControl(BatControl.class);
}

private BallControl getBallControl() {
    return getGameWorld().getSingleton(BreakoutType.BALL).get().getControl(BallControl.class);
}

protected void onCollisionBegin(Entity ball, Entity brick) {
    brick.getControl(BrickControl.class).onHit();
}
```

BreakoutApp sets up methods referred to BallControl, BatControl, BrickControl types.

BallControl, BatControl, BrickControl still have meanings (be used by other classes) if BreakoutApp is deleted.

(5) BallControl, BatControl, BrickControl are parts of BreakoutFactory (Aggregation)

```
return Entities.builder()
        .from(data)
        .type(BreakoutType.BAT)
        .at(FXGL.getSettings().g
        .viewFromNodeWithBBox(F
        .with(physics, new Colli
        .with(new BatControl())
```

```
return Entities.builder()
        .from(data)
        .type(BreakoutType.BALL)
        .bbox(new HitBox("Main",
        .viewFromNode(new Circle
        .with(physics, new Colli
        .with(new BallControl(),
```

```
return Entities.builder()
        .from(data)
        .type(BreakoutType.BRICK)
        .viewFromNodeWithBBox(FXGl
        .with(new PhysicsComponent
        .with(new BrickControl())
```

BreakoutFactory returns values referred to BallControl, BatControl, BrickControl types.

BallControl, BatControl, BrickControl still have meanings (be used by other classes) if BreakoutFactory is deleted.

(6) BreakoutApp owns BreakoutFactory (Composition)

```
TextLevelParser parser = new TextLevelParser(new BreakoutFactory());
```

```
public static void main(String[] args) {
    launch(args);
}
```

BreakoutApp is the main class of the game (an executable file to launch the game). It is responsible for the life cycle of the BreakoutFactory object.

(7) BreakoutApp, BreakoutFactory uses BreakoutType (Greneral form of dependency relationship)

```
private BatControl getBatControl() {
    return getGameWorld().getSingleton(BreakoutType.BAT).get().getControl(BatControl.class);
}

private BallControl getBallControl() {
    return getGameWorld().getSingleton(BreakoutType.BALL).get().getControl(BallControl.class);
}
```

```
getPhysicsWorld().addCollisionHandler(new CollisionHandler(BreakoutType.BALL, BreakoutType.BRICK) {
.type(BreakoutType.BRICK)
.type(BreakoutType.BAT)
.type(BreakoutType.BALL)
```

getPhysicsWorld().addCollisionHandler(new CollisionHandler(BreakoutType.BALL, BreakoutType.BRICK) {

BreakoutApp and BreakoutFactory use BreakoutType to simplify specifying types.

After understanding codes, it is time to improve codes into an acceptable standard form (The design pattern will be introduced in the good code example).

1. Code Formatting

```
@Override
protected void initSettings(GameSettings settings) {
    settings.setTitle("Breakout Underwater");
    settings.setVersion("0.2");
       settings.setWidth(600);
          settings.setHeight(800);
    settings.setIntroEnabled(false);
         settings.setMenuEnabled(false);
                settings.setProfilingEnabled(false);
    settings.setCloseConfirmation(false);
    settings.setApplicationMode(ApplicationMode.DEVELOPER);
}
```

class BreakoutApp Line78~89

This is a typical example of bad code formatting. Though there are no strict rules for code formatting and this example code can be compiled successfully, it will result in great trouble of maintaining and extending these codes for other programmers.

Indentation, newlines, uppercase and lowercase, all these constitutes hierarchy of codes. A good hierarchy of codes can make codes more readable and easy to rebuild.

A good format of code should be like this:

```
@Override
protected void initSettings(GameSettings settings) {
    settings.setTitle("Breakout Underwater");
    settings.setVersion("0.2");
    settings.setWidth(600);
    settings.setHeight(800);
    settings.setIntroEnabled(false);
    settings.setMenuEnabled(false);
    settings.setProfilingEnabled(false);
    settings.setCloseConfirmation(false);
    settings.setApplicationMode(ApplicationMode.DEVELOPER);
}
```

2. Variable Naming

```
private PhysicsComponent g2;
```

class BallControl Line 39

Identifiers are not only a variable name, but also a signal showing what the variable does. A good identifier will not only help developers avoid some unnecessary Javadoc comments (John 2001), but also help those programmers who later do maintenance and extension more easily.

In this example, 'g2' is really a confusing variable name and is never used in the rest part of the class. The developer may want to name the variable as 'physics' (which is used below).

A good variable naming should be like this (from my point of view):

```
private PhysicsComponent physics;
```

3. Setting Specifiers

```
public static final float BOUNCE_FACTOR = 1.5f;
public static final float SPEED_DECAY = 0.66f;

public GameEntity bat;
public PhysicsComponent physics;
public float speed = 0;

public Vec2 velocity = new Vec2();
```

class BatControl Line 41~48

Specifiers are used to set properties of member variables. These properties include access authority, whether it is static or not, whether it is a constant.

In this example, all variables created here are only used in BatControl class. Therefore, programmers should change their specifiers from 'public' to 'private', to protect these variables from being changed by another class or method which is not in the same class.

'Private' variables implement encapsulation principle perfectly. They can only be accessed by access interface which is designed by programmers. Therefore, they have a characteristic of high security.

A good specifiers setting should be like this:

```
private static final float BOUNCE_FACTOR = 1.5f;
private static final float SPEED_DECAY = 0.66f;

private GameEntity bat;
private PhysicsComponent physics;
private float speed = 0;

private Vec2 velocity = new Vec2();
```

4. Implementing Comment format

```
// we add IrremovableComponent because regardless of the level
// the background and screen bounds stay in the game world
Entities.builder()
        .viewFromNode(bg)
        .with(new IrremovableComponent())
        .buildAndAttach(getGameWorld());

Entity screenBounds = Entities.makeScreenBounds(40);
screenBounds.addComponent(new IrremovableComponent());
```

class BreakoutApp Line 145~153

```
/*
if (bat.getX() < 0) {
    velocity.set(BOUNCE_FACTOR * (float) -bat.getX(), 0);
} else if (bat.getRightX() > FXGL.getApp().getWidth()) {
    velocity.set(BOUNCE_FACTOR * (float) -(bat.getRightX() - FXGL.getApp().getWidth()), 0);
}*/
```

class BatControl Line 67~72

Implementation comments are often used for commenting out code or for comments about the particular implementation.

In the first example, there are two lines of comments. Therefore, it should follow the block comment format, since '//' comment delimiter should not be used on consecutive multiple lines for text comments. However, it can be used in consecutive multiple lines for commenting out sections of code. Therefore, the second example also does a bad performance (Sun Microsystems, Inc 1997).

A good comment format should be like this:
First one:

```
/* we add IrremovableComponent because regardless of the level
 * the background and screen bounds stay in the game world
 */
Entities.builder()
        .viewFromNode(bg)
        .with(new IrremovableComponent())
        .buildAndAttach(getGameWorld());

Entity screenBounds = Entities.makeScreenBounds(40);
screenBounds.addComponent(new IrremovableComponent());
```

Second one:

```
//if (bat.getX() < 0) {
//    velocity.set(BOUNCE_FACTOR * (float) -bat.getX(), 0);
//} else if (bat.getRightX() > FXGL.getApp().getWidth()) {
//    velocity.set(BOUNCE_FACTOR * (float) -(bat.getRightX() - FXGL.getApp().getWidth()), 0);
//}
```

After showing examples of bad codes, there are also some good codes in this 'Break-out' game which deserve learning.

5. Object-Oriented Design Simple Factory Pattern (Creational)

```
public class BreakoutFactory implements TextEntityFactory {

    @SpawnSymbol('1')
    public Entity newBrick(SpawnData data) {
        return Entities.builder()
                .from(data)
                .type(BreakoutType.BRICK)
                .viewFromNodeWithBBox(FXGL.getAssetLoader().loadTexture("brick_blue.png", 232 / 3, 104 / 3))
                .with(new PhysicsComponent(), new CollidableComponent(true))
                .with(new BrickControl())
                .build();
    }

    @SpawnSymbol('9')
    public Entity newBat(SpawnData data) {
        PhysicsComponent physics = new PhysicsComponent();
        physics.setBodyType(BodyType.KINEMATIC);

        return Entities.builder()
                .from(data)
                .type(BreakoutType.BAT)
                .at(FXGL.getSettings().getWidth() / 2 - 50, 30)
                .viewFromNodeWithBBox(FXGL.getAssetLoader().loadTexture("bat.png", 464 / 3, 102 / 3))
                .with(physics, new CollidableComponent(true))
                .with(new BatControl())
                .build();
    }

    @SpawnSymbol('2')
    public Entity newBall(SpawnData data) {
        PhysicsComponent physics = new PhysicsComponent();
          physics.setBodyType(BodyType.DYNAMIC);
            physics.setFixtureDef(new FixtureDef().restitution(1f).density(0.03f));

               ParticleEmitter emitter = ParticleEmitters.newFireEmitter();


               emitter.numParticles = 5;
               emitter.setEmissionRate = 0.5 ;

        return Entities.builder()
                .from(data)
                .type(BreakoutType.BALL)
                .bbox(new HitBox("Main", BoundingShape.circle(10)))
                .viewFromNode(new Circle(10, Color.LIGHTCORAL))
                .with(physics, new CollidableComponent(true))
                .with(new BallControl(), new ParticleControl(emitter))
                .build();
    }
```

class BreakoutFactory Line55~101

Simple factory pattern, also known as static factory method pattern, is to define a specific method which is responsible for creating other classes' instances. These instances usually have the same super class.

Therefore, whenever you need an instance, what you need to do is to pass a correct parameter to the factory method and you do not need to know the details of creating the instance.

Take this 'break-out' game as an example, I will illustrate why this factory pattern is a good object-oriented design pattern.

Firstly, BreakoutApp (Client) is on longer responsible for creating objects, it only cares about using objects. The pattern realizes the separation of object's creation and use.

Secondly, client does not need to know the detailed information of objects, it only needs to know the corresponding parameter to each object.

Finally, this OO design pattern can improve program flexibility by using configuration files to adapt different situations.

Conclusion

From the bad code examples above, we can see that these bad codes are really small points that are usually ignored by programmers. Just as the old saying goes, Rome was not built in a day. We need to pay attention to this problem as early as possible. To develop a good habit before we get into actual work. Hence, we can develop good and secure codes which are easy for maintenance and extension.

In conclusion, making codes good is a crucial part of developing software, which deserves all programmers' attention, and later, it will be got to be worth paying for.

Reference List

Design Pattern - Factory Pattern (no date). Available from https://www.tutorialspoint.com/design_pattern/factory_pattern.htm [7th Nov]

John, F. (2001) Make bad code good. Available from https://www.javaworld.com/article/2075129/testing-debugging/make-bad-code-good.html [4th Nov 2017]

Kathy, S., Bert, B. (2005) Head First Java. O'Reilly Media

Robert, M. (2006) Abstract Factory by Daniel T. Available from http://butunclebob.com/ArticleS.UncleBob.AbstractFactoryDanielT [6th Nov 2017]

Ron, J. (2017) Clean Code: A Learning. Available from https://ronjeffries.com/articles/017-08ff/clean-code/ [7th Nov 2017]

Sun Microsystems, Inc (1997) Java Code Conventions. Available from http://www.oracle.com/technetwork/java/codeconventions-150003.pdf [5th Nov 2017]