# P2: Exercise Session 8

Claudio Corrodi

# Coding Issues: Attributes

```java
class Board {
    public Square firstSquare;
}

private void client() {
    Square start = board.firstSquare;
    // ...
}
```

# Coding Issues: Attributes

```java
class Board {
    public Square firstSquare;
}

private void client() {
    Square start = board.firstSquare;
    // ...
}
```

What if we change “firstSquare”?

# Coding Issues: Attributes

```
class Board {
    public List<Square> squares;
}

private void client() {
    Square start = squares.get(0);
    // ...
}
```

Does not work anymore! We need to change code in all clients!

What if we change “firstSquare”?

# Coding Issues: Attributes

```
class Board {
    private Square firstSquare;

    public Square getFirstSquare() {
        return firstSquare;
    }
    public void setFirstSquare(Square aSquare) {
        firstSquare = aSquare;
    }
}

private void client() {
    Square start = board.getFirstSquare();
    // …
}
```

With getters/setters, we can change the implementation without affecting clients.

# Coding Issues: Attributes

```
class Board {
    private List<Square> squares;

    public Square getFirstSquare() {
        return squares.get(0);
    }
    public void setFirstSquare(Square aSquare) {
        squares.set(0, aSquare);
    }
}

private void client() {
    Square start = board.getFirstSquare();
    // …
}
```

With getters/setters, we can change the implementation without affecting clients.

# Coding Issues: Attributes

- Make attributes private
  - Rarely protected, almost never package or public visibility

- Use getters and setters to make them available
  - Getter without setter: Read only value
  - Can always increase complexity of getters and setters, don't have to expose data structure

# Coding Issues: Constants

```java
public class Board {
    private final int BOARD_SIZE;
    private final char[] ROW_NAMES = { 'A', 'B', 'C' };
    private final int[] COL_NAMES = { 1, 2, 3 };
}
```

# Coding Issues: Constants

```
public class Board {
    private final int BOARD_SIZE;
    private final char[] ROW_NAMES = { 'A', 'B', 'C' };
    private final int[] COL_NAMES = { 1, 2, 3 };
}
```

These are not constants!

# Coding Issues: Constants

```java
public class Board {
    private final int BOARD_SIZE;
    private final char[] ROW_NAMES = { 'A', 'B', 'C' };
    private final int[] COL_NAMES = { 1, 2, 3 };
}
```

These are not constants!

```java
public class Board {
    private final int boardSize;
    private final char[] rowNames = { 'A', 'B', 'C' };
    private final int[] colNames = { 1, 2, 3 };
}
```

Use CamleCase for attributes

# Coding Issues: Constants

```
public class Board {
    private final int BOARD_SIZE;
    private final char[] ROW_NAMES = { 'A', 'B', 'C' };
    private final int[] COL_NAMES = { 1, 2, 3 };
}
```

These are not constants!

```
public class Board {
    private final int boardSize;
    private final char[] rowNames = { 'A', 'B', 'C' };
    private final int[] colNames = { 1, 2, 3 };
}
```

Use CamleCase for attributes

```
public class Board {
    private static final int BOARD_SIZE = 3;
    private static final char[] ROW_NAMES = { 'A', 'B', 'C' };
    private static final int[] COL_NAMES = { 1, 2, 3 };
}
```

‘static final’ for constants

# Coding Issues: Constants vs enumerations

```
final class Direction {
    public static final int LEFT = 1;
    public static final int RIGHT = 2;
    public static final int UP = 3;
    public static final int DOWN = 4;
}
```

```
public static Command createCommand(int type) {
    if (type == LEFT) {
        return new CommandLeft();
    } else if (type == RIGHT) {
        return new CommandRight();
    } else {
        // ...
    }
    return null;
}
```

# Coding Issues: Constants vs enumerations

```
final class Direction {
    public static final int LEFT = 1;
    public static final int RIGHT = 2;
    public static final int UP = 3;
    public static final int DOWN = 4;
}
```

```
public static Command createCommand(int type) {
    if (type == LEFT) {
        return new CommandLeft();
    } else if (type == RIGHT) {
        return new CommandRight();
    } else {
        // ...
    }
    return null;
}
```

Lots of “if-then-else” statements. Code smell!

# Coding Issues: Constants vs enumerations

```
enum Direction {
    LEFT,
    RIGHT,
    UP,
    DOWN
}
```

```
Command createCommand(Direction dir) {
    switch (dir) {
        case LEFT: return new CommandLeft();
        case RIGHT: return new CommandRight();
        case UP: // ...
        case DOWN: // ...
    }
    // ...
}
```

Slightly better, less error prone.

# Coding Issues: Constants vs enumerations

```java
interface CommandFactory {
    Command create();
}
enum Direction implements CommandFactory {
    LEFT {
        public Command create() {
            return new CommandLeft();
        }
    },
    RIGHT {
        public Command create() {
            return new CommandRight();
        }
    },
    // ...
}
```

Enums can implement interfaces

# Coding Issues: Constants vs enumerations

```
interface CommandFactory {

}
er
```

```
// Client
Command createCommand(Direction dir) {
    return dir.create();
}
```

```
          }
    },
    // ...
}
```

Enums can implement interfaces

# Coding Issues: Switch instructions

```
private int convertToInt(char c) {
    int output;
    switch (c) {
        case 'a': output = 0;
        case 'b': output = 1;
        case 'c': output = 2;
        case 'd': output = 3;
        case 'e': output = 4;
        case 'f': output = 5;
        case 'g': output = 6;
        case 'h': output = 7;
        case 'i': output = 8;
        case 'j': output = 9;
        default: output = 10;
    }
    return output;
}
```

What does convertToInt(‘e’) return?

# Coding Issues: Switch instructions

```
private int convertToInt(char c) {
    int output;
    switch (c) {
        case 'a': output = 0;
        case 'b': output = 1;
        case 'c': output = 2;
        case 'd': output = 3;
        case 'e': output = 4;
        case 'f': output = 5;
        case 'g': output = 6;
        case 'h': output = 7;
        case 'i': output = 8;
        case 'j': output = 9;
        default: output = 10;
    }
    return output;
}
```

Always prints 10!

What does convertToInt(‘e’) return?

# Coding Issues: Switch instructions

```java
private int convertToInt(char c) {
    int output;
    switch (c) {
        case 'a': output = 0; break;
        case 'b': output = 1; break;
        case 'c': output = 2; break;
        case 'd': output = 3; break;
        case 'e': output = 4; break;
        case 'f': output = 5; break;
        case 'g': output = 6; break;
        case 'h': output = 7; break;
        case 'i': output = 8; break;
        case 'j': output = 9; break;
        default: output = 10; break;
    }
    return output;
}
```

Don’t forget to break or return!

# Coding Issues: Switch instructions

```java
private boolean isLowercaseLetterBeforeE(char c) {
    boolean result;
    switch (c) {
        case 'a':
        case 'b':
        case 'c':
        case 'd':
            result = true;
            break;
        default:
            result = false;
            break;
    }
    return result;
}
```

“Falling through” can be useful…

# Coding Issues: Switch instructions

```java
public boolean isLowercaseLetterBeforeE(char c) {
    return c - 'a' < 4;
}
```

This is a bit simpler

# Coding Issues: Switch instructions

```
public boolean isLowercaseLetterBeforeE(char c) {
    return c - 'a' < 4;
}
```

But is it a good implementation?

# Coding Issues: Switch instructions

```java
public boolean isLowercaseLetterBeforeE(char c) {
    assert c >= 'a' && c <= 'z';
    return c - 'a' < 4;
}
```

Better?

# Coding Issues: Switch instructions

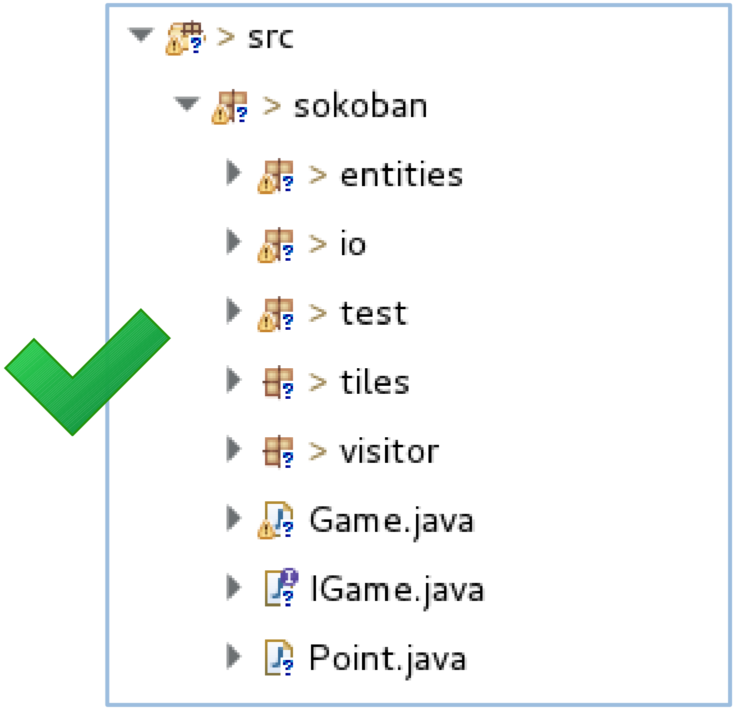
```java
/**
 * Checks whether the given character comes
 * before 'e' in the alphabet.
 * @param c a character, must be a lowercase
 *          letter between 'a' and 'z'
 */
public boolean isLowercaseLetterBeforeE(char c) {
    assert c >= 'a' && c <= 'z';
    return c - 'a' < 4;
}
```

Don’t forget your contracts!

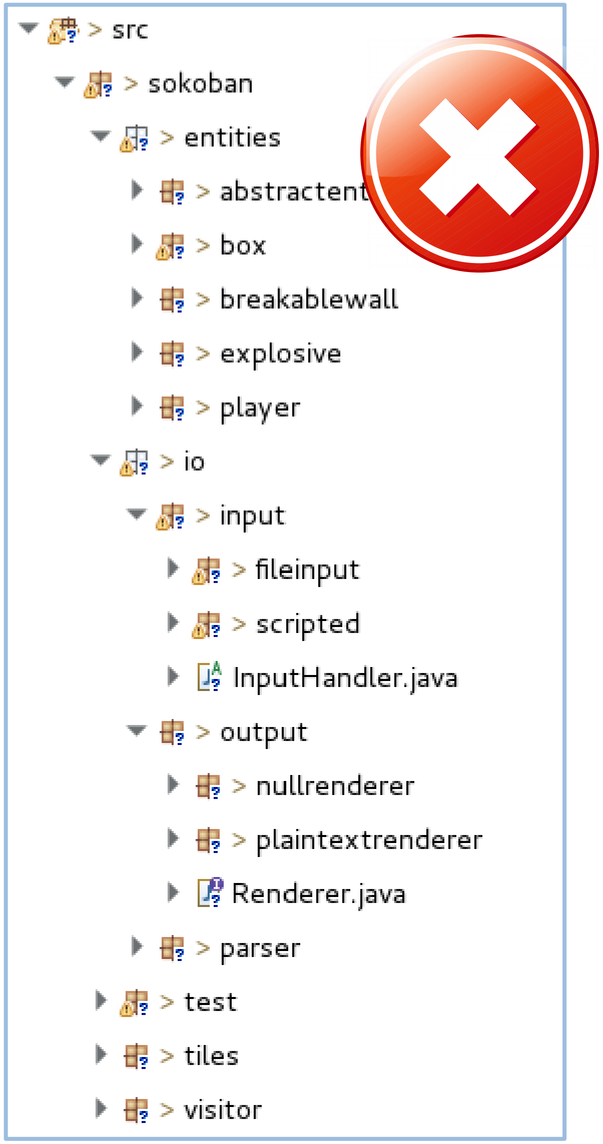
# Coding Issues: Packages

- Don’t overdo it! You don’t need one package per class.

- Start with few packages. You can always easily refactor (drag and drop in Eclipse).



```
> src
  > sokoban
    > entities
    > io
    > test
    > tiles
    > visitor
    Game.java
    IGame.java
    Point.java
```



```
> src
  > sokoban
    > entities
      > abstractent
      > box
      > breakablewall
      > explosive
      > player
    > io
      > input
        > fileinput
        > scripted
        InputHandler.java
      > output
        > nullrenderer
        > plaintextrenderer
        Renderer.java
      > parser
    > test
    > tiles
    > visitor
```

# Sketching

"A sketch is a **rapidly executed** freehand **drawing** that is not usually intended as a finished work."

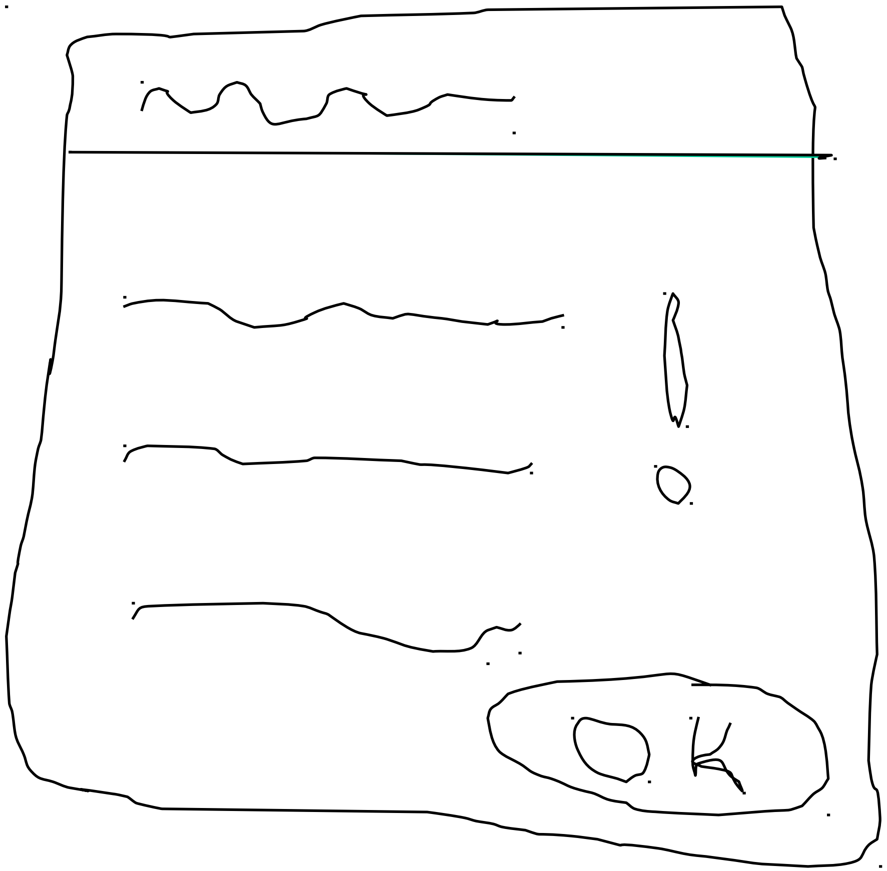


Figure from the slides on "Sketching User Experiences: The Workbook"

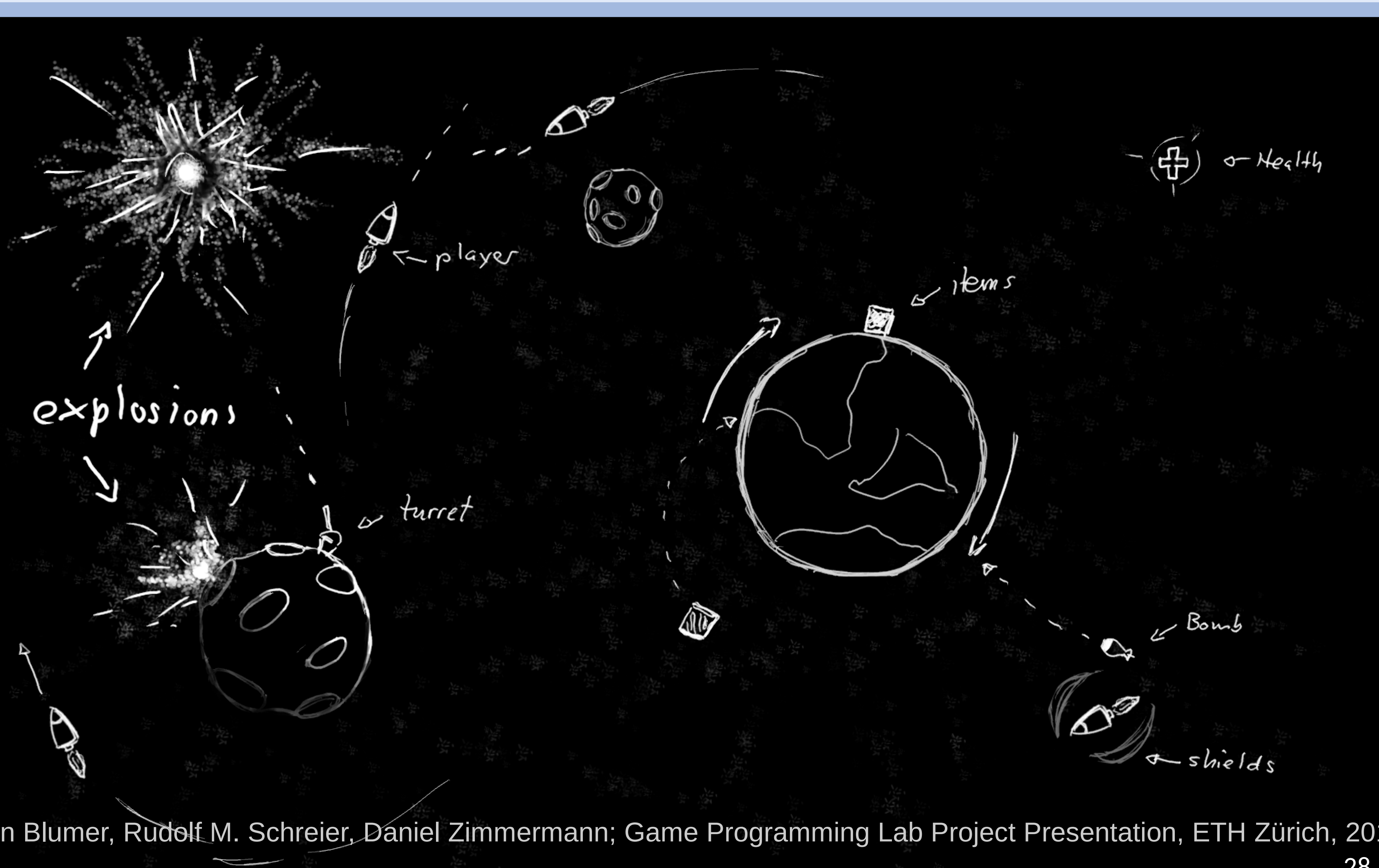
https://en.wikipedia.org/wiki/Sketch_%28drawing%29

# Sketching

- Sketching is helps you to
  - express
  - develop, and
  - communicate design ideas
- Force yourself to visualize how things come together
- Brainstorming
  - Come up with as many ideas as possible
  - It is about design, not function
- Quick to make, disposable, plentifulb

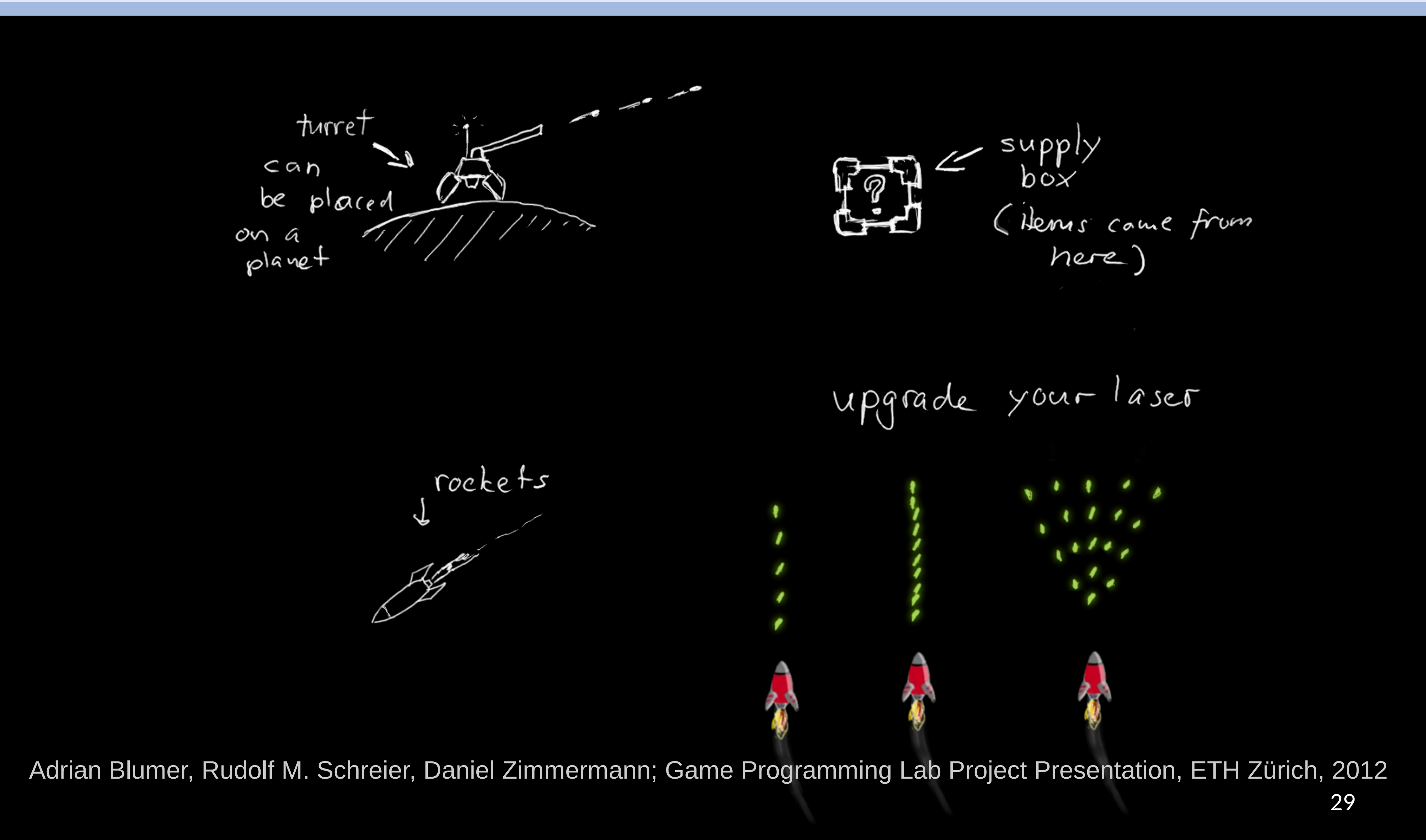
Adapted from the slides on “Sketching User Experiences: The Workbook”

# Sketching a game



Adrian Blumer, Rudolf M. Schreier, Daniel Zimmermann; Game Programming Lab Project Presentation, ETH Zürich, 2012

# Sketching a game



Adrian Blumer, Rudolf M. Schreier, Daniel Zimmermann; Game Programming Lab Project Presentation, ETH Zürich, 2012

# Physical prototyping

- A physical prototype consists of
  - A set of objects and sketches that resemble the intended user interface
  - A set of rules (how can a player move? What actions are allowed in what state?)

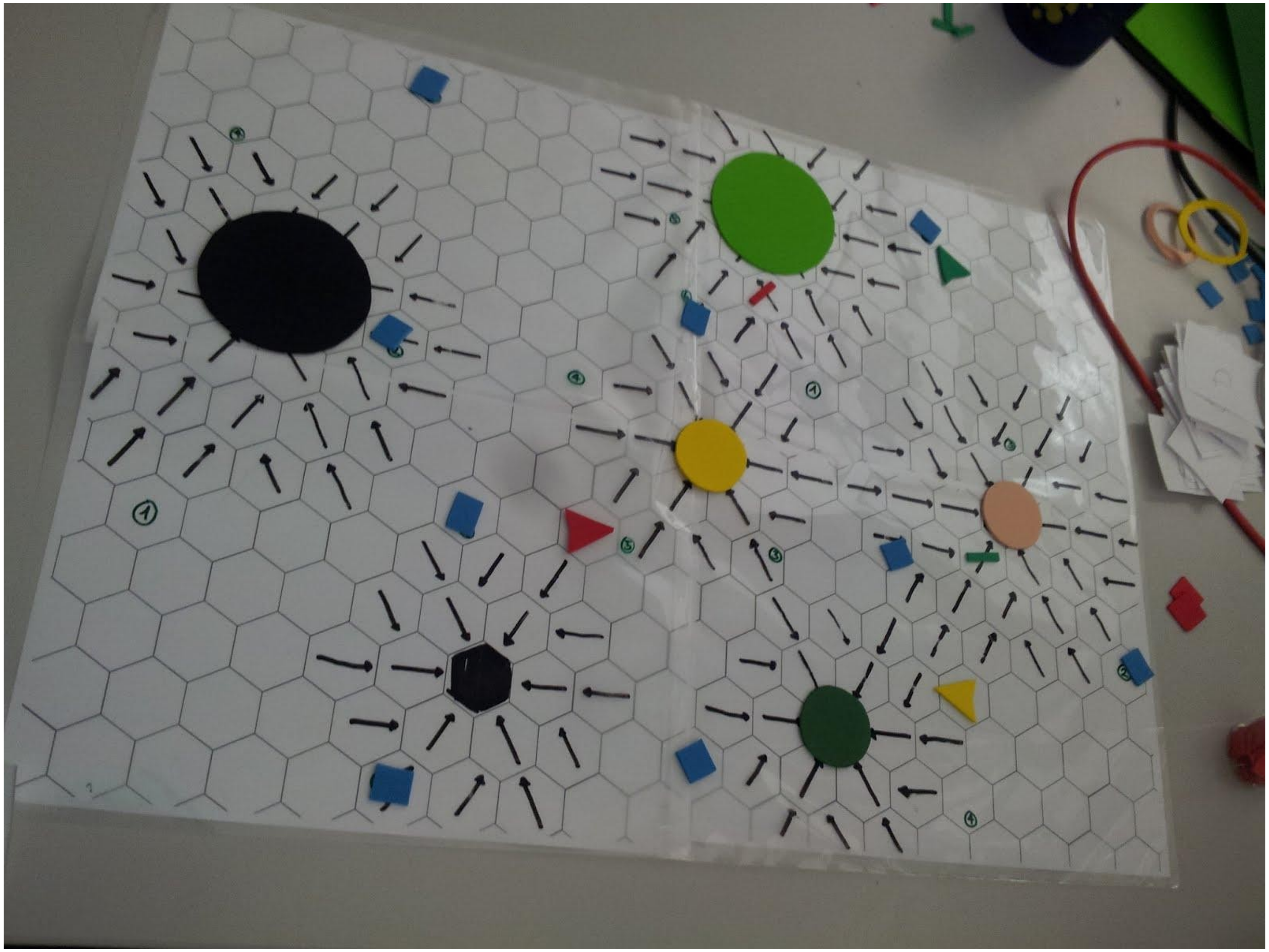
- and lets you
  - Simulate the user experience by executing rules and moving the elements on the board

- Should focus on the core elements

- Can be developed iteratively to refine the design

Based on: https://graphics.ethz.ch/teaching/gamelab11/course_material/lecture03/Prototyping.pdf

# Paper Prototype



# Resources

- Slides on “Sketching User Experiences: The Workbook”

  http://sketchbook.cpsc.ucalgary.ca/?page_id=64

- Adrian Blumer, Rudolf M. Schreier, Daniel Zimmermann; Game Programming Lab Project Presentation, ETH Zürich, 2012

  https://twiki.graphics.ethz.ch/GameClass/Team2

- Game Programming Laboratory Course Notes, CGL Group, ETH Zürich

  https://graphics.ethz.ch/teaching/gamelab16/notes.php

# Exercise 8

- Sketch Sokoban interfaces
  - Pen & Paper
  - No programming required
- Create a simple paper prototype
  - Demonstrate user experience
- Improve unit and integration tests
  - Consequent use of mocking to test individual methods
  - Integration tests for exercising multiple classes / the whole system

# Evaluation Form

- Evaluation form available

  https://docs.google.com/forms/d/1ADYZ-ECF2vzuECI8rC7C_f-1dIoL2nfyCVDA8MsdmAM/viewform

  (the link is also in exercise_08.md and on Piazza)

- Anonymous and not part of Exercise 8
  - But we would appreciate your participation!