Conceptual Modeling and Programming in GIScience

Lecture 5: Introduction

Yingjing Huang

yingjing.huang@univie.ac.at

2026S – May 11, 2026

What You Will Learn in This Lecture

- Take **polymorphism** seriously, not as a label
- Learn what an ***interface*** is and how it differs from an abstract class
- Learn to combine multiple interfaces on a single class
- Learn when to reach for an interface vs an abstract class
- Learn what a class invariant is and how ***private*** plus delegation protects it
- Learn to read and draw a **UML class diagram**
- Meet cellular automata and build **Conway's Game of Life**
- Learn **2D arrays**, nested loops, and the simultaneous-update pattern

Polymorphism

Dynamic dispatch picks the right *getArea()* at runtime based on the **actual object** type

```
for (Geometry g : list) {
    total += g.getArea();   // dispatches to each shape's own version
}
```

Abstract Class

A parent that cannot be instantiated, forcing subclasses to provide implementations

```
public abstract class Geometry {
    public abstract double getArea();   // every subclass must fill in
}
```

Composition (has-a)

The list is *private*; all access is gated through PolyLine's own methods to protect **Invariants**

```
public class PolyLine {
    private ArrayList<Point> points;     // PolyLine HAS-A ArrayList
}
```

ArrayList

Dynamic, generic collections.

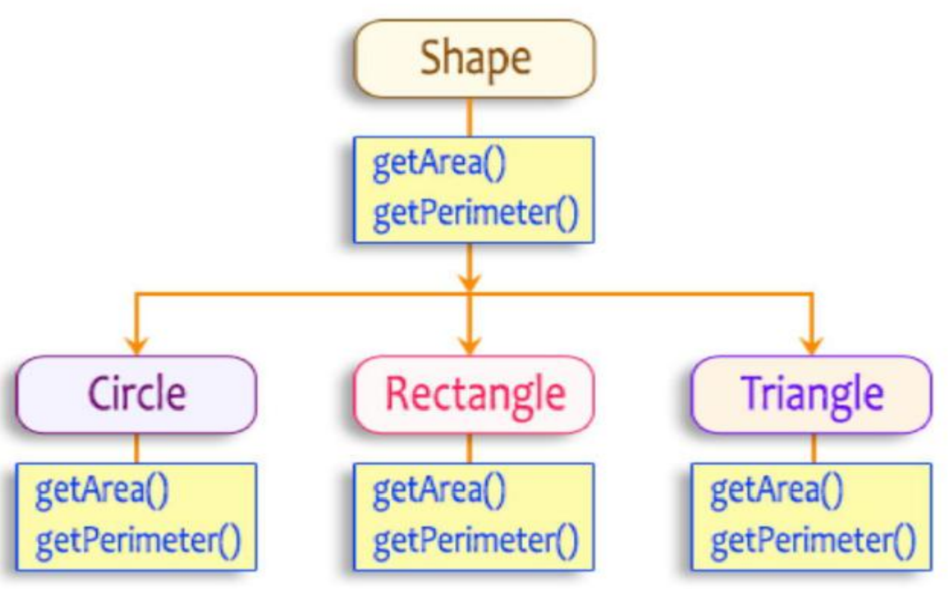
Today: We transition to fixed-size 2D arrays for Grid simulations

```
ArrayList<Person> population = new ArrayList<>();
population.add(new Person());
```

Polymorphism: Many Forms

poly = many **morph** = form

- Variable declared as general type (e.g., *Shape*)
- Holds specific objects (e.g., *Circle*, *Rectangle*)
- Method calls run the **actual object's** version



One name on the page, many forms at runtime.

Manual Dispatch

```
for (Shape s : shapes) {
    if (s instanceof Circle c)         c.drawCircle();
    else if (s instanceof Rectangle r) r.drawRectangle();
    else if (s instanceof Triangle t)  t.drawTriangle();
}
```

Adding a new shape requires editing every loop like this.

Polymorphic Dispatch

```
for (Shape s : shapes) {
    s.draw();
}
```

Open to extension, closed to modification.

Add Triangle? Just write *Triangle.draw()* once.

1 Single Inheritance

A class can only extend **one** parent.

What if a Shape needs to be Drawable AND Comparable?

```
class Circle extends Shape, Drawable, Comparable
 // ❌ won't compile
```

2 Pure Contracts

Sometimes there is zero shared state. An abstract class with no fields and only signatures is just a contract pretending to be a class.

```
public abstract class Drawable {
    public abstract void draw();       // no body
    public abstract Box bounds();      // no body
    // no fields, no concrete methods — what's
left of "class"?
}
```

What an interface is

- A pure contract: a list of method signatures with no bodies.
- Any class that **claims** to implement it must provide every method.

The Metaphor

A **Job Description** lists required skills. Multiple different people can fill the role in their own way. The employer (calling code) only needs the description.

```java
public interface Drawable {
    void draw();                    // no body, ends with ;
    Box  bounds();                  // signatures only
    void select(boolean on);
}
```

- Keyword *interface* replaces *class*
- Capability names: *Drawable*, *Comparable*, *Iterable*
- Methods are implicitly *public* and *abstract*

```java
public class Circle implements Drawable {
    private double radius;

    @Override
    public void draw()     { /* draw a circle */ }

    @Override
    public Box bounds()    { /* return bounding box */ }

    @Override
    public void select(boolean on) { /* toggle highlight */ }
}
```

- *implements* signs the class up for the contract
- The compiler **refuses** the class until every interface method is implemented
- *@Override* works on interface methods too — same protection against typos
- Once implemented, a *Circle* can be used anywhere a *Drawable* is expected

	extends	***implements***
What	class inherits from another class (is-a relationship)	Class fulfills an interface contract (acts-as relationship)
Inherits fields?	yes	no – interfaces have none
Inherits bodies?	yes – can be overridden	no – class writes its own
How many?	exactly **one** parent	as many as you needed
Keyword to call up	*super*	(not applicable)

```
class Circle extends Shape implements Drawable, Comparable<Circle> { ... }
```

Composing Multiple Capabilities

```
public class Point extends Geometry
        implements Drawable, Comparable<Point>, Serializable { ... }
```

- One class can implement **as many interfaces as you needed**
- Each interface contributes **one capability**
- No "diamond problem"
  - — interfaces have no implementation to clash, only signatures
- The price: **you write each method's implementation yourself**
- The reward: **clean type hierarchy, free composition of capabilities**

```java
public interface Drawable {
    void draw();                          // still abstract – no body

    default Box bounds() {                // default method – has a body
        return Box.unitBox();             // implementers get this for free
    }
}
```

Why Java 8+?

- Add methods to old interfaces without breaking legacy code

  e.g., *forEach* in *List*

- Provide sensible fallback behavior
- **Warning**: Don’t abuse. If you need fields, use an abstract class.

Three quick rules

- **Real shared implementation** (fields + method bodies)

  → **abstract class**. Interfaces can't hold state
- **Pure capability cutting across unrelated types**

  → **interface**. Drawable applies to shapes, icons, labels, widgets equally
- **Behavior from multiple sources**

  → **interface(s)**. You only get to extend one parent

When in doubt: interface first.

It's more flexible and forces you to think in contracts.

An invariant is a promise the class makes about itself, **always**.

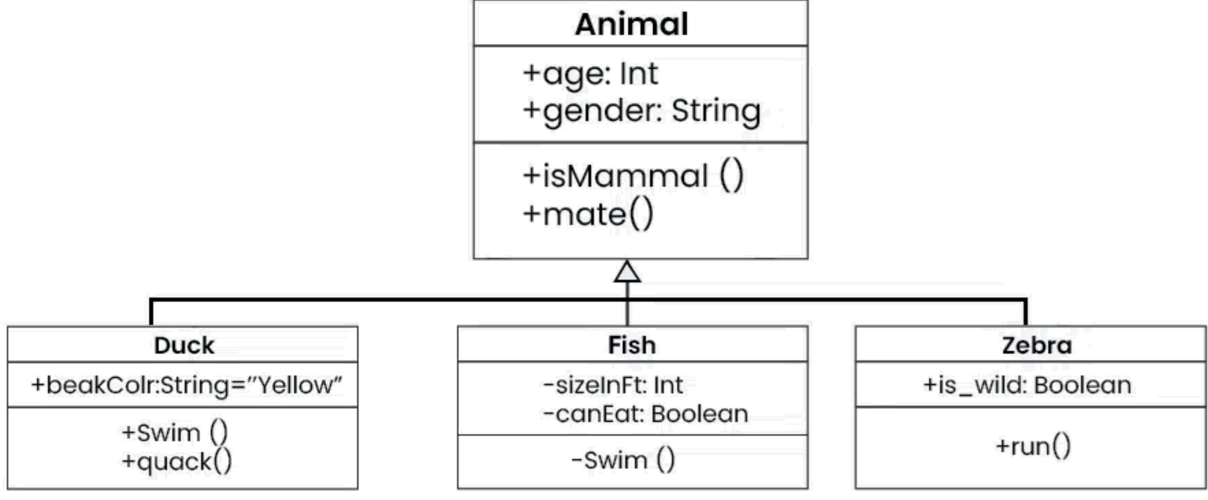
```
public class PolyLine {
    private ArrayList<Point> points;      // private – outside cannot touch
    private double length;                // cached total

    public void add(Point p) {
        if (!points.isEmpty()) {
            length += points.get(points.size() - 1).distanceTo(p);
        }
        points.add(p);                    // single gate – list and length stay in sync
    }
}
```

For PolyLine: "My stored *length* always equals the actual sum of segment lengths."

Whenever one piece of state must stay consistent with another, you need an invariant — and private fields to protect it.

UML – Unified Modeling Language

- See **all** your types in one picture instead of jumping between files
- **Communicate** a design to a teammate before either of you writes code
- **Plan** a refactor — try alternatives on the diagram, not in the codebase



UML Class Diagram Essentials

ClassName
- privateField : type + publicField : type
+ method(arg: type) : type

Class Box

Top: class name (*italic* for interface or abstract class)

Middle: fields, prefixed with visibility

Bottom: methods, same visibility prefixes

Visibility Markers

+ Public

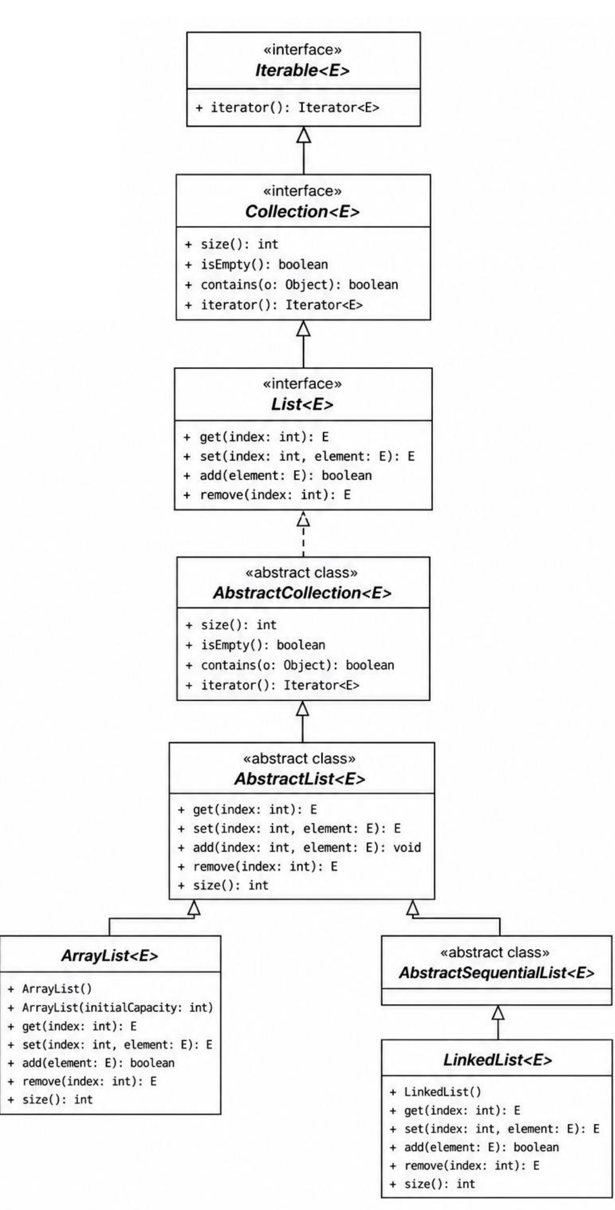
Private

Protected

Arrows

Relationship	Line	Head
Inheritance (*extends*)	solid	hollow triangle
Implementation (*implements*)	dashed	hollow triangle
Association (has-a, uses)	solid	open arrow
Dependency (temporary use)	dashed	open arrow

Java Collections as a Class Diagram



The chain underneath every

import java.util.ArrayList

First half — structure

how to organize types and relationships

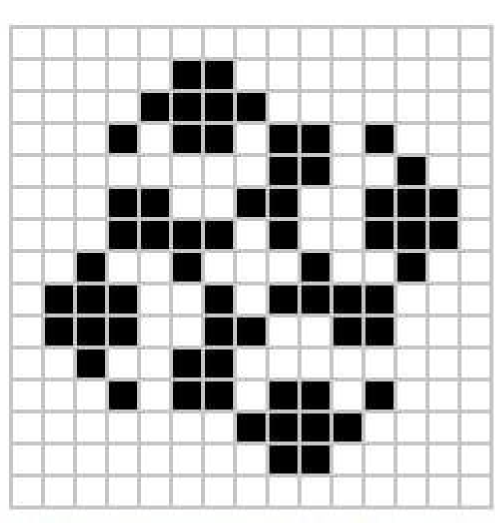
(interfaces, invariants, UML)

Second half — dynamics

how state evolves over discrete time

(cellular automata, Game of Life, the simultaneous-update pattern)

- A **grid** of cells
- Each cell is in one of a **finite set of states** (in the simplest case, alive / dead)
- Time advances in **discrete steps**
- At each step, every cell's **next state** is determined by **fixed local rules** based on its current state and its neighbors
- **All cells update simultaneously**



Conway's Game of Life: Four Rules

Under-pop

Live cell with **< 2** neighbors dies.

Survival

Live cell with **2 or 3** neighbors lives.

Over-pop

Live cell with **> 3** neighbors dies.

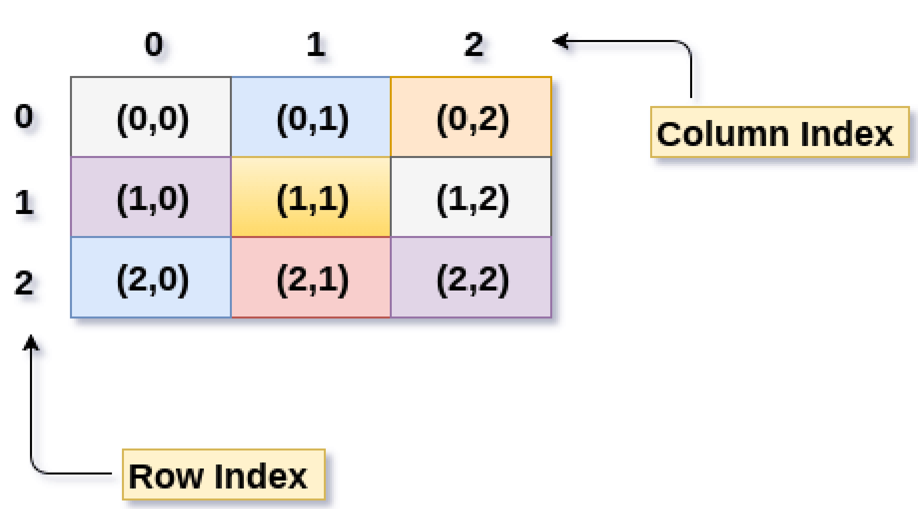
Reproduction

Dead cell with **= 3** neighbors revives.

Representing the Grid: 2D Arrays

```java
boolean[][] grid = new boolean[ROWS][COLS];

// read a cell
boolean alive = grid[i][j];

// write a cell
grid[i][j] = true;
```

- Array of Arrays
- Fixed size at creation
- Memory efficient for grids



Iterating Through the Grid

```
for (int i = 0; i < ROWS; i++) {
    for (int j = 0; j < COLS; j++) {
        // visit cell (i, j) exactly once
        process(grid[i][j]);
    }
}
```

- Outer loop walks the **rows (i)**
- Inner loop walks the **columns (j)**
- Always use distinct names: i and j
- Each pass of the inner loop visits exactly **one cell**

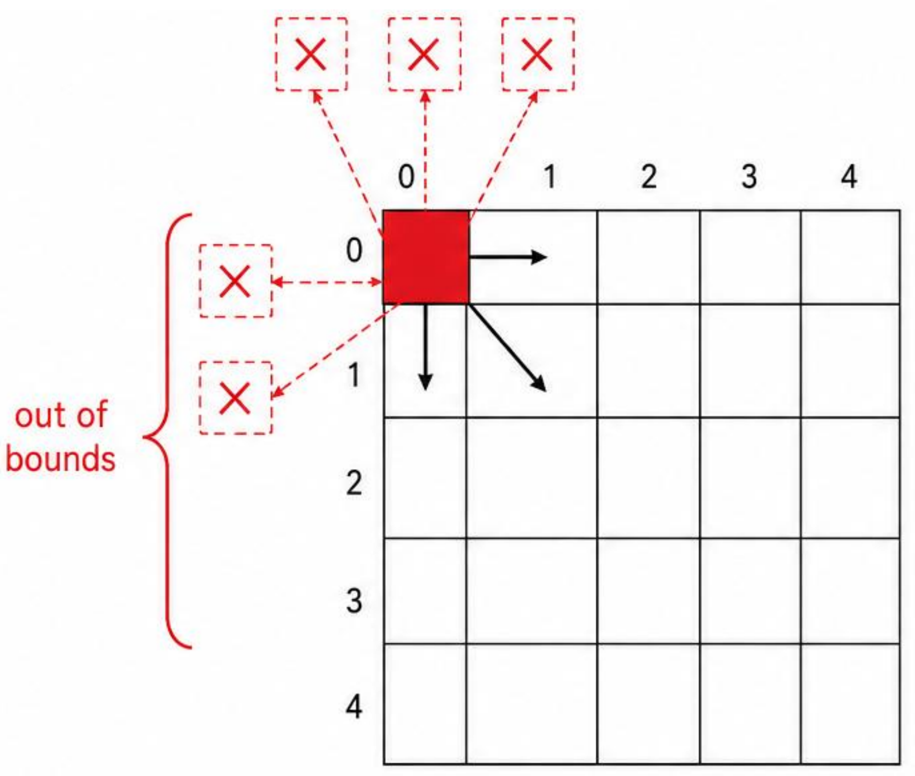
```
int countLiveNeighbours(int i, int j) {
    int count = 0;
    for (int di = -1; di <= 1; di++) {
        for (int dj = -1; dj <= 1; dj++) {
            if (di == 0 && dj == 0) continue;        // skip self
            int ni = i + di, nj = j + dj;
            if (inBounds(ni, nj) && grid[ni][nj]) {
                count++;
            }
        }
    }
    return count;
}
```

- Three-by-three window around (i, j), skip the center.
- Eight neighbors: N, S, E, W, NE, NW, SE, SW (queen's-move adjacency)
- inBounds(ni, nj) guards against going off the edge

```
private boolean inBounds(int ni, int nj) {
    return ni >= 0 && ni < ROWS && nj >= 0 && nj < COLS;
}
```

Don't let your code crash with

ArrayIndexOutOfBoundsException



The Simultaneous-Update Trap

The rules say: *every cell's next state depends on the* ***current*** *state of all neighbors*

If you write each new state back into the same grid as you walk it:

```
// ❌ wrong
for (int i = 0; i < ROWS; i++) {
    for (int j = 0; j < COLS; j++) {
        grid[i][j] = nextStateOf(i, j);     // overwrites — corrupts later reads
    }
}
```

- By the time cell (0,1) is processed, cell (0,0) already changed
- Cell (0,1) reads the **new** (0,0) — not the **current** one
- Output looks plausible but **isn't Game of Life anymore**

Solution: Copy-and-Swap

```
public void tick() {
    boolean[][] next = new boolean[ROWS][COLS];   // ① fresh

    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
            next[i][j] = nextStateOf(i, j);       // ② read grid, write next
        }
    }

    grid = next;                                  // ③ swap — single assignment
}
```

- grid is **untouched** during the whole computation
- next is filled in based on grid's current state — consistent, by construction
- One assignment at the end commits the new generation

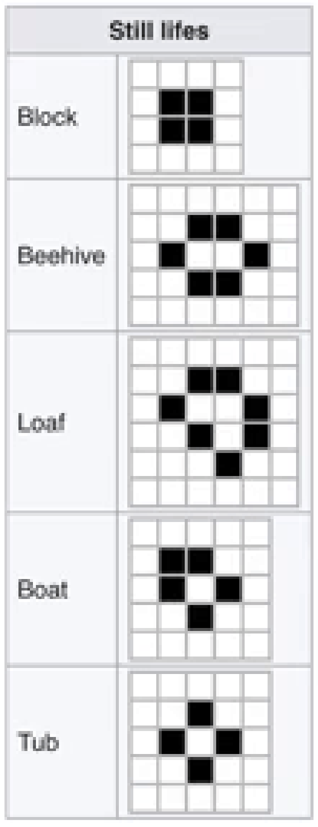
Other names for the same idea:

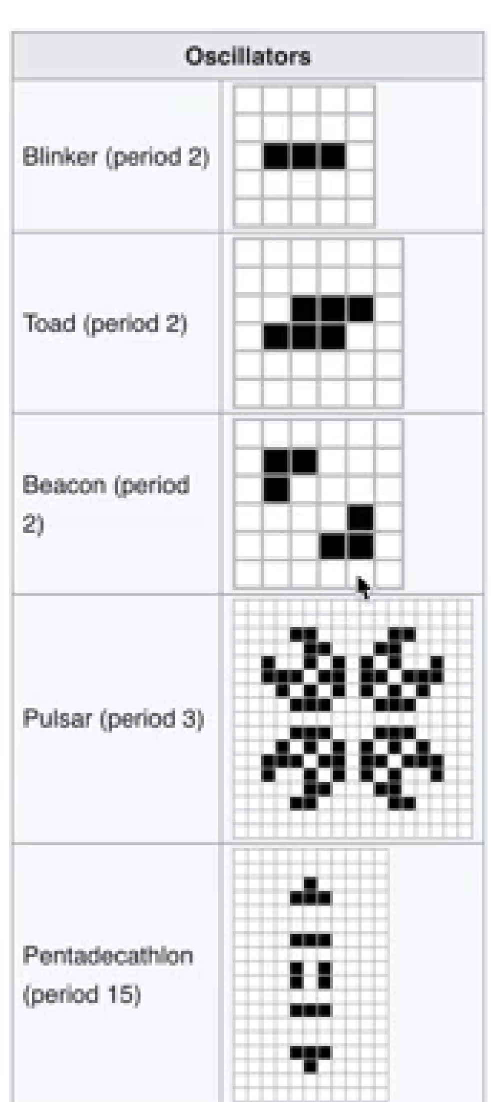
double-buffering (graphics), immutable update (functional programming).

```
while (true) {
    System.out.println(world);                    // ① print current grid
    world.tick();                                 // ② advance one step
    Thread.sleep(500);                            // ③ pause so you can see it
}
```

- **Render**: Override *toString()* to show # (alive) and . (dead)
- **Pause**: 500ms allows the human eye to follow patterns

  For debugging, slow it down to 1500-2000ms so you can follow each step
- **Exceptions**: Thread.sleep requires a try-catch or throwing *InterruptedException*





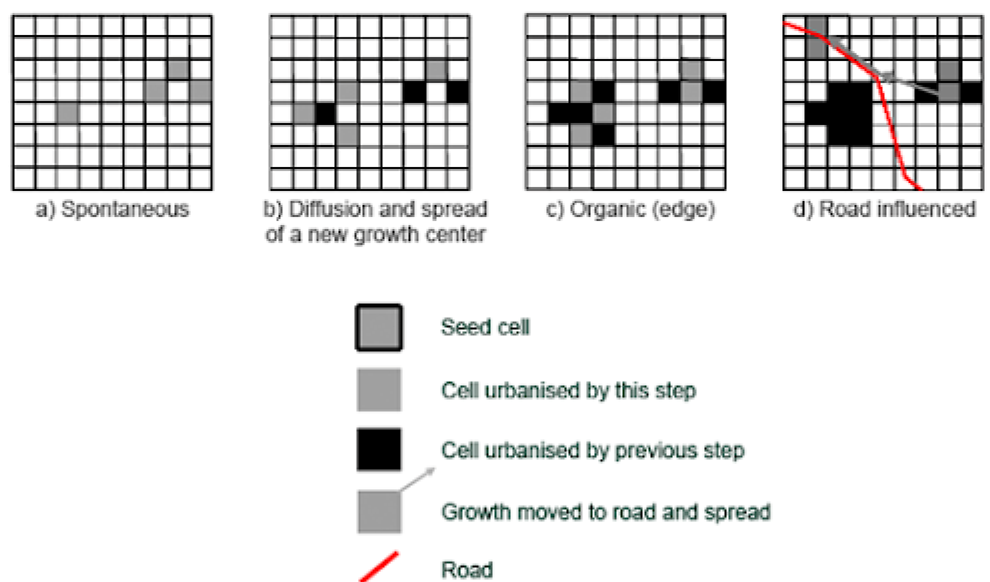
Spaceships	
Glider	
Lightweight spaceship (LWSS)	
Middleweight spaceship (MWSS)	
Heavyweight spaceship (HWSS)	

***emergence** in complex systems*

Simple local rules → Complex global behavior

Cellular Automata in Geography

- **Urban growth** — Clarke's SLEUTH model: cells transition between *undeveloped / developed*, rules calibrated from historical data
- **Land-use change** — forest → farmland → suburb, driven by neighborhood pressure and accessibility
- **Wildfire and flood propagation** — local spread rules over terrain rasters
- **Disease spread** — your SIR work fits exactly here



Reading: Read the book chapters on interfaces and 2D arrays

Coding

- **Drawable + Comparable on Geometry** in Code Example. Add a *Drawable* interface with *String draw()*. Make *Circle* and *Rectangle* implement *Drawable* and *Comparable<Geometry>* (by **area**). In *main*, sort a mixed list with *Collections.sort* and print each via *draw()*.
- **Finish the Game of Life template** in Code Example. Fill in *tick()*, *neighbours()*, toString(). Test on the blinker.
- **Fix the SIR step with copy-and-swap.** Refactor your W4 code, so every person's next state is computed from the population's current state, then applied at the end.

Submission: FirstNameLastNameW5.zip (*.java files only)

Deadline: Sunday (before W7) at 5:00 PM

Midterm Scope

	Contents (might included but not limited)
Week 1	integer arithmetic, loop counts, primitive vs. reference types, …
Week 2	encapsulation, ‘this’, constructors, pass-by-value, …
Week 3	‘extends’, implicit ‘Object’, ‘==’, ‘static’, ‘final’, ‘ArrayList’, …
Week 4	‘super(…)’, dynamic dispatch, ‘instanceof’, abstract classes, ‘@Override’, seeded ‘Random’, …
Week 5	interfaces, ‘implements’, polymorphism, copy-and-swap, ..

What you will not be asked in the exam:

(but it’s always good to know these)

- UML drawing
- Game of Life implementation
- Zombie corridor
- OGC standard and WKT Representation

Midterm Scope

The goal of this course is **not** to train everyone to become professional software developers.

The more important objective is

- Master the **translation** of real world challenges into code
- Build the intuition to choose **appropriate modeling tools**

Good luck with the midterm :)

Lab Time