CS 4414: Recitation 13

Sagar Jha

Clarification on vector capacity (Thanks to Chris Gyurgyik)

- I simplified things last time by saying that capacity is doubled every time a vector needs more space
- C++ standard, in fact, does not specify how capacity should grow with the size
- According to https://tylerayoung.com/2020/08/20/default-capacity-growth-rate-of-c-stdvector/, it grows at "something vaguely resembling exponential"
- Capacity exactly doubles with GCC 5.1, GCC 10.2, Clang 6, and Clang 10.0.1 (sequence 0, 1, 2, 4, 8, …). I tested this on Linux.
- With MSVC 2013 and 2019, it follows the sequence:
  0, 1, 2, 3, 4, 6, 9, 13…

Have you ever struggled with writing code?

Have you ever struggled with writing code?

- Found yourself clueless at the start?
- Found that the code became too complicated too soon?
- Found yourself unable to deal with the many bugs?
- Did not enjoy the coding process?

Then this recitation is for you!

```
DisplayObject nest[4] = {
DisplayObject("\
\\     /#\
 -------", 0),

DisplayObject("\
\\  O  /#\
 -------", 0),
...
};

DisplayObject chicken("\
     O>#\
   ^( )#\
    = =", 2);

DisplayObject cow("\
   ____ \"#\
 /(____)U#\
  !! !!", 3);

DisplayObject farmer("\
   _#\
  O#\
/(~)\\#\
 ! !", 1);
```

```
barn.draw(1, 1);
bakery.draw(10, 50);
cow.draw(17, 18);
farmer.draw(22, 19);
child.draw(30, 19);
eggs.draw(14, 43);
flour.draw(18, 43);
sugar.draw(22, 43);
butter.draw(26, 43);
truck.draw(42, 15);
```

```
for(int n = 0; n < 10000; n++)
  {
    nest[n % 4].draw(10, 10);
    cupcakes[n % 7].draw(15, 80);
    ...
      mixer_contents.update_contents(mixer_string);
      mixer_contents.draw(26, 72);
      batter.draw(19, 74);
      baked = true;
    ...
      y = std::max(1, y + (1+std::rand()) % 10 - 5);
      x = std::max(1, x + (1+std::rand()) % 10 - 5);
      chicken.draw(oldy = y, oldx = x);
```

Motivation: What’s wrong with the above code?

```
DisplayObject nest[4] = {
DisplayObject("\
\\      /#\
 -------", 0),

DisplayObject("\
\\  0  /#\
 -------", 0),
...
};

DisplayObject chicken("\
  O>#\
 ^( )#\
  = =", 2);

DisplayObject cow("\
   ___\"#\
 /(____)U#\
  !! !!", 3);

DisplayObject farmer("\
   _#\
  O#\
/(~)\\#\
 ! !", 1);
```

```
barn.draw(1, 1);
bakery.draw(10, 50);
cow.draw(17, 18);
farmer.draw(22, 19);
child.draw(30, 19);
eggs.draw(14, 43);
flour.draw(18, 43);
sugar.draw(22, 43);
butter.draw(26, 43);
truck.draw(42, 15);
```

```
for(int n = 0; n < 10000; n++)
  {
    nest[n % 4].draw(10, 10);
    cupcakes[n % 7].draw(15, 80);
    ...
      mixer_contents.update_[illegible](mixer_string);
      mixer_contents.draw(26, 72);
      batter.draw(19, 74);
      baked = true;
    ...
      y = std::max(1, y + (1+std::rand()) % 10 - 5);
      x = std::max(1, x + (1+std::rand()) % 10 - 5);
      chicken.draw(oldy = y, oldx = x);
```

Everything is a display object at the same level, defined in main

Positions of objects are hard-coded

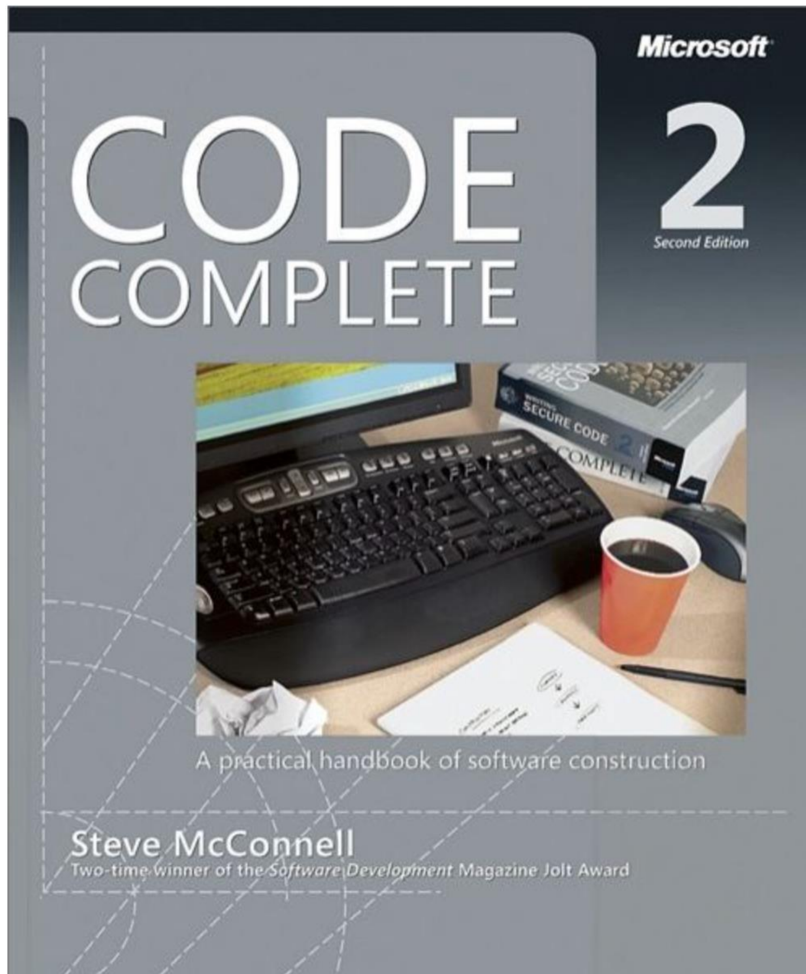
Movement is hard-coded, not independent, and state variables are mixed in the code

Motivation: What's wrong with the above code?

Today: How to write better code

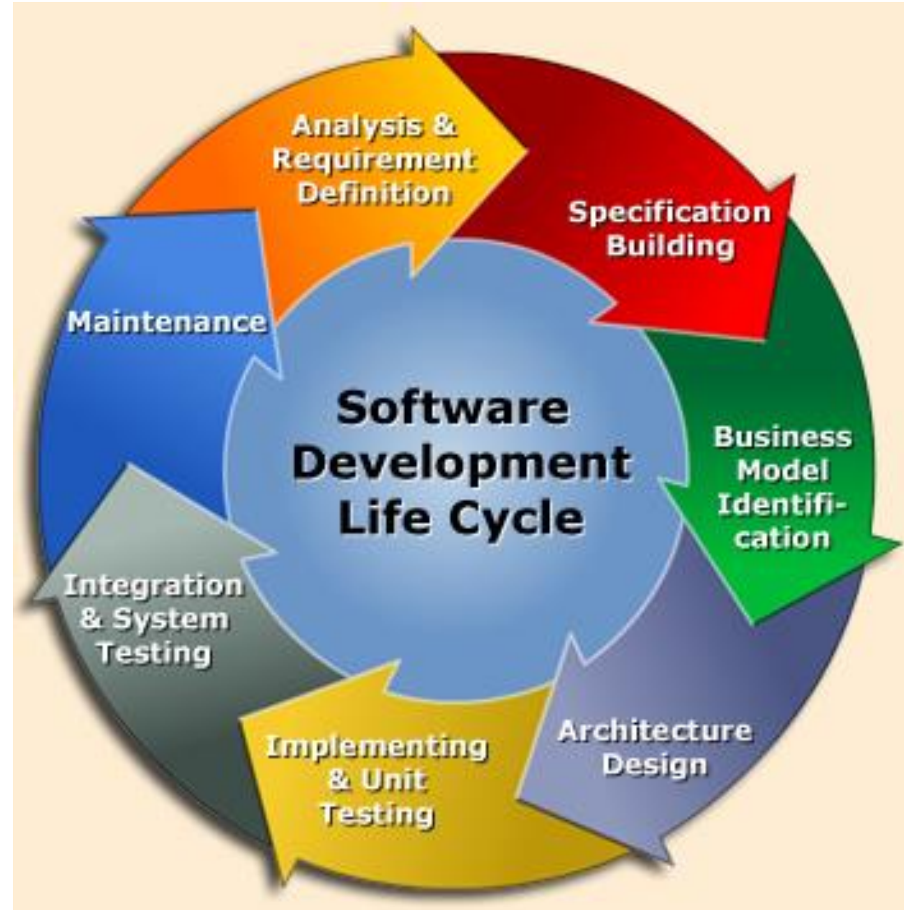
Focus on software design

- Understanding software design
- Identify goals of code development
- Understanding the different levels of design
- Designing subsystems/packages, individual classes, and their routines
- Learning how to identify higher forms of abstraction
- Working through the implementation details

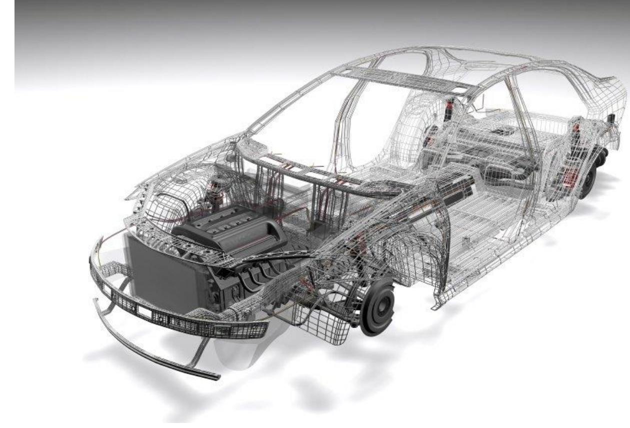


Reference: Code Complete, 2nd Edition
Part II: Creating High-Quality Code

Software development process



What is software design?



- Planning a solution for a problem before writing code
- Clarifying/exploring the specifications and the constraints
- Identifying the goals – performance requirements, scope of the solution, reusability, portability
- Designing subsystems, classes, and class routines

Features of software design

- Design is hard: Sometimes, the requirements are known only by solving partially. Specifications, constraints and requirements change constantly while development is going on
- Design requires use of heuristics: Trying well-known patterns, or designing by trial and error
- Understand your goals before beginning to design
- Design incrementally: Come up with a first design, develop some features, test, then add more features

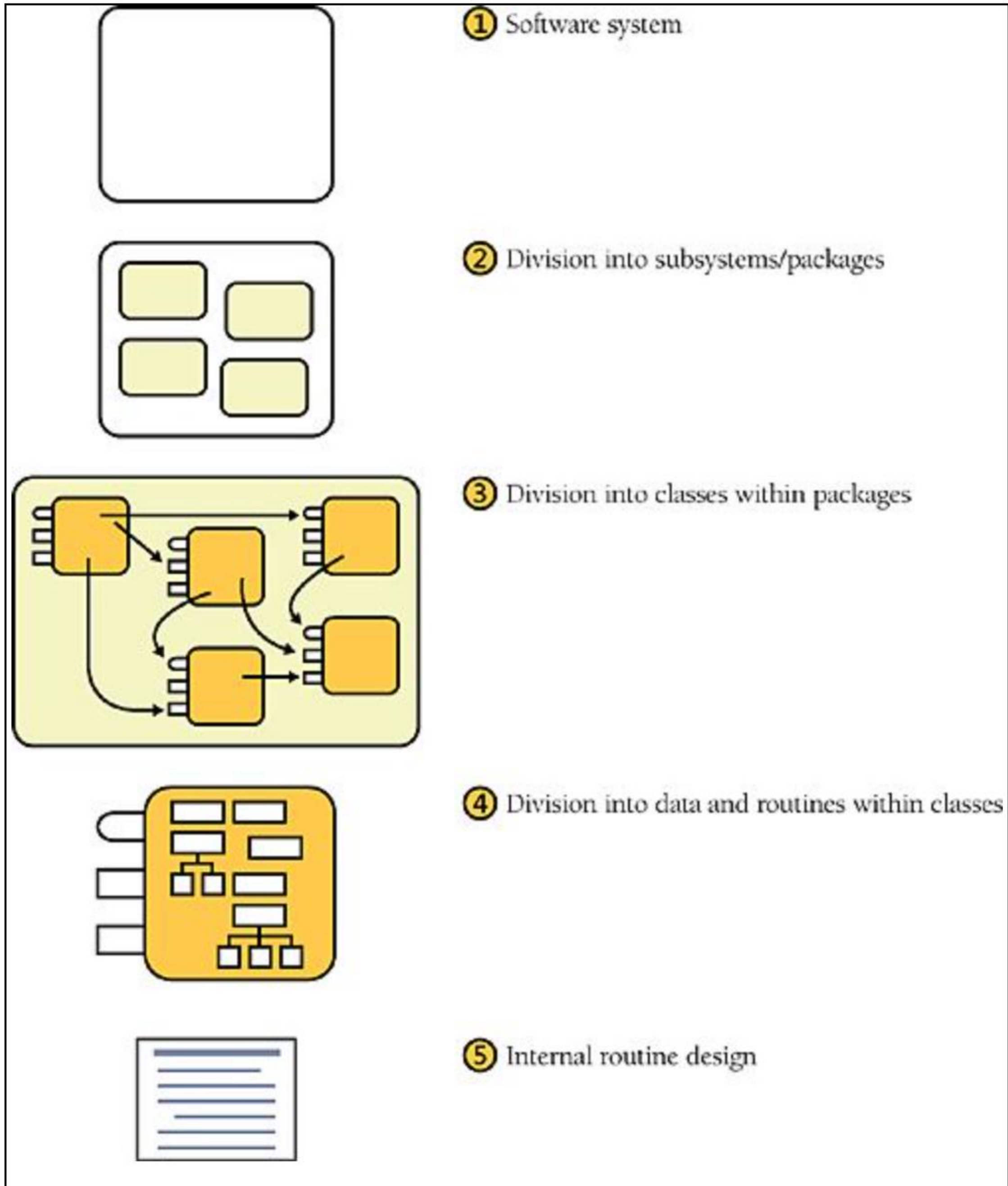
Developing Farmville: Goals

- Achieve a clean separation between objects, their movement, the state of the world, and the display
- Start with very few objects. Make it easier to add more
- Localize how objects interact with each other and how they avoid collisions
- Avoid coarse-grained locks over the entire world state
- Encode movement speed and the interaction between draw and display efficiently
- Make it easier to change collision avoidance logic, size of the world, display logic etc.

One of the most important goals: **Managing complexity**

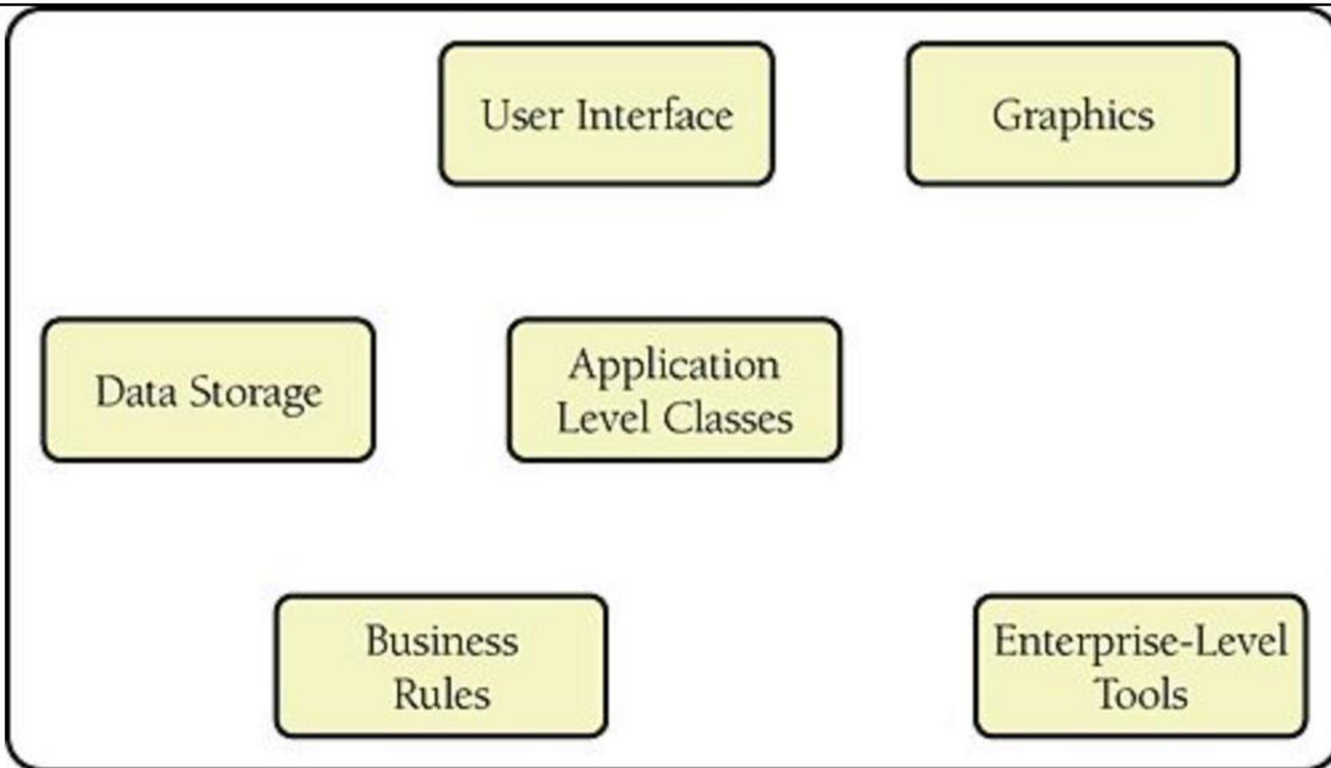
Break a complicated problem into simple pieces:

- Divide systems into subsystems
- Separation of concerns: Carefully define classes/objects with clear boundaries
- Keep functions (routines) short
- Work at the highest level of abstraction
- Focus on simplicity and ease-of-understanding
- Extensibility
- Loose coupling (plug-and-play)



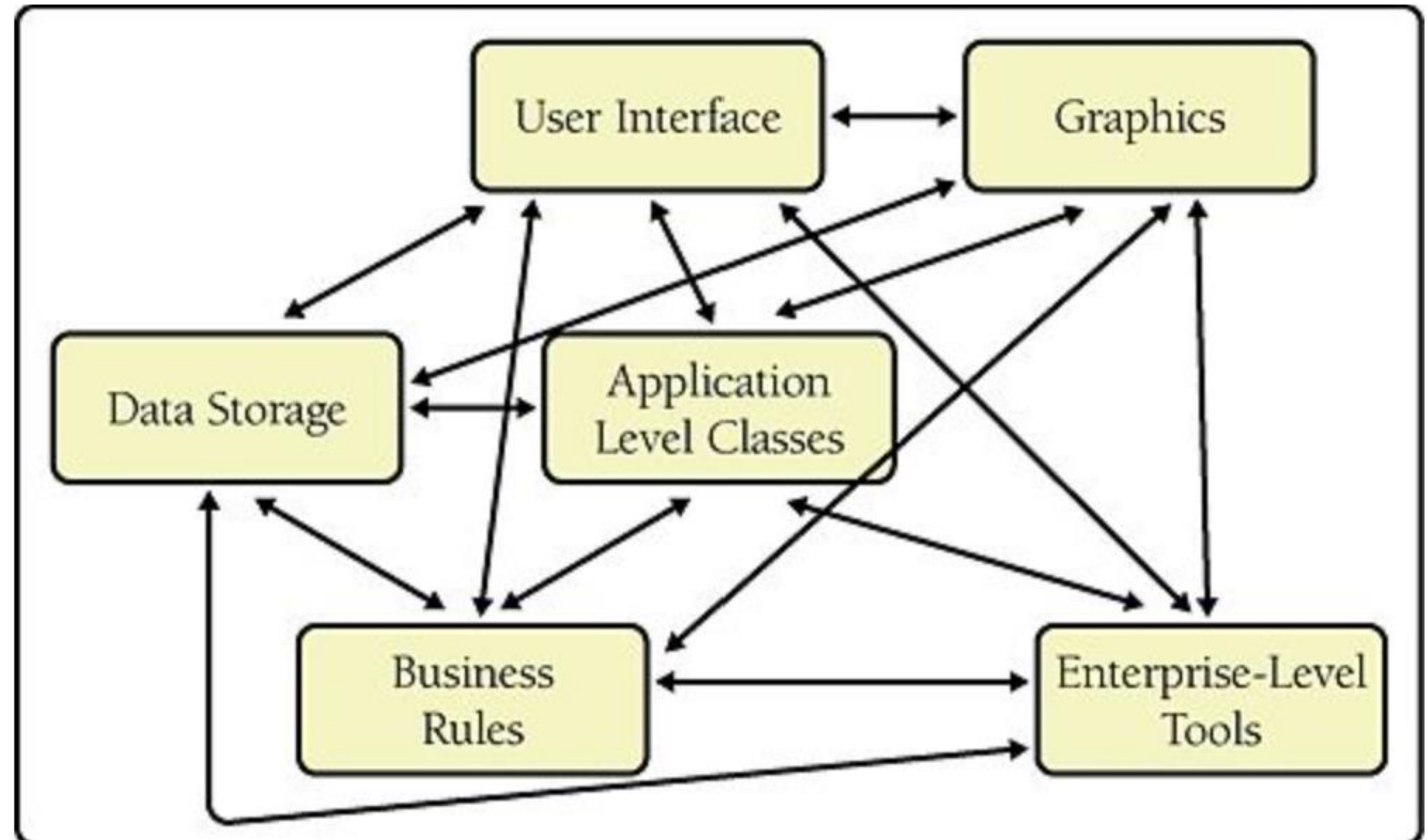
Levels of design

Level 2: Division into subsystems/packages

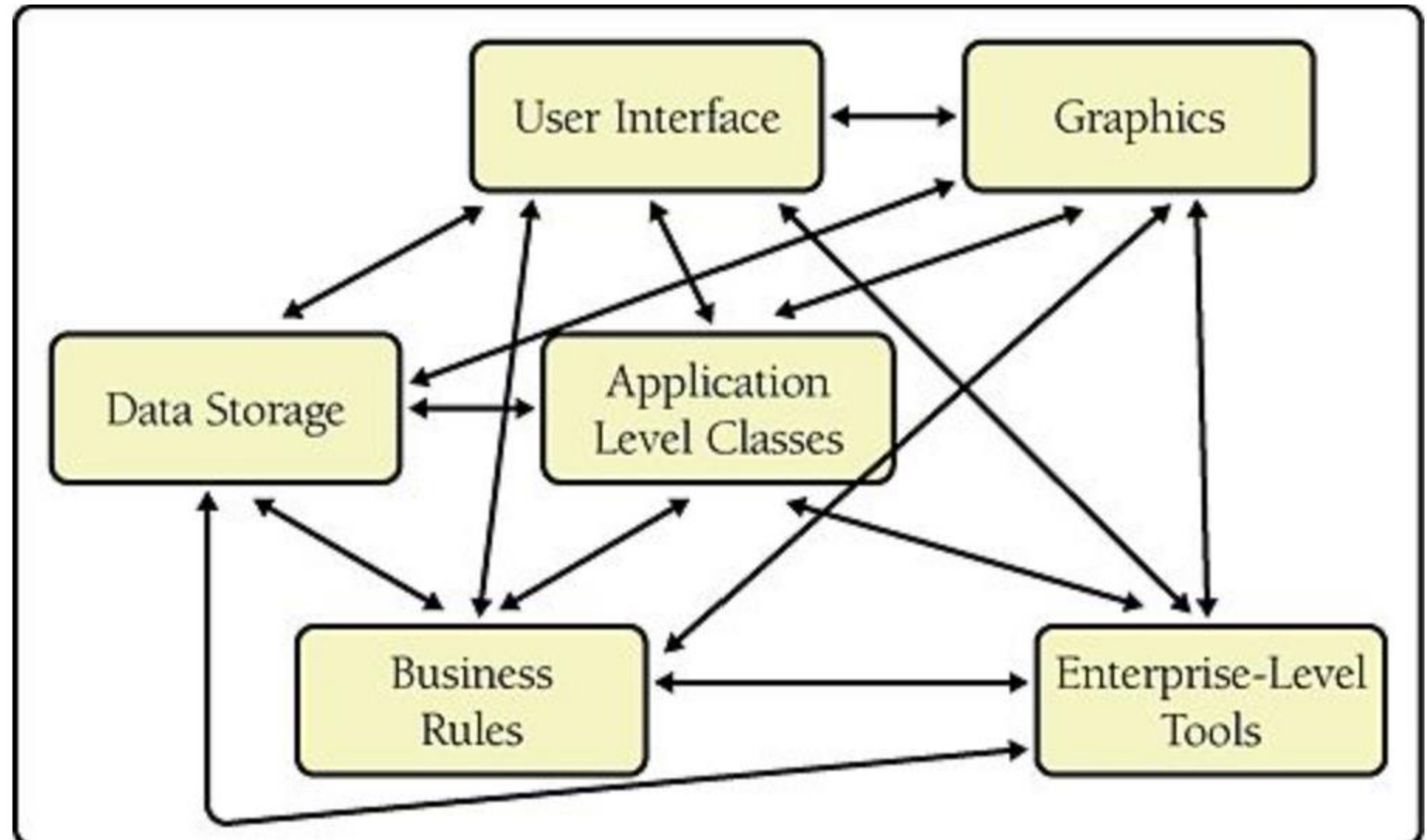


- Subsystem: Inter-related collection of classes
- Try grouping based on common sense
- Revise if necessary
- Define clear interfaces for each subsystem
- Define how subsystems interact with each other

What's wrong with the following communication pattern?

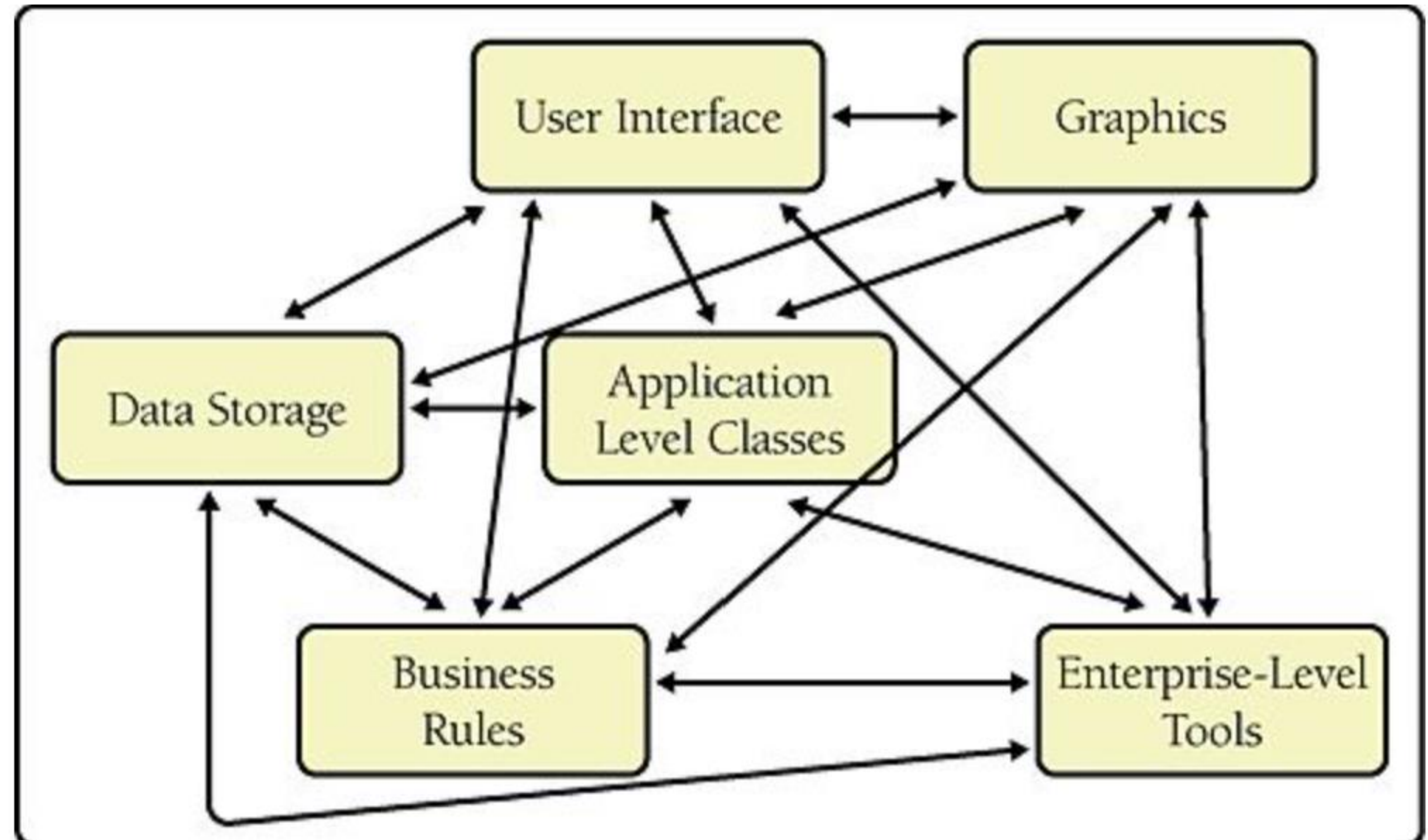


What’s wrong with the following communication pattern?



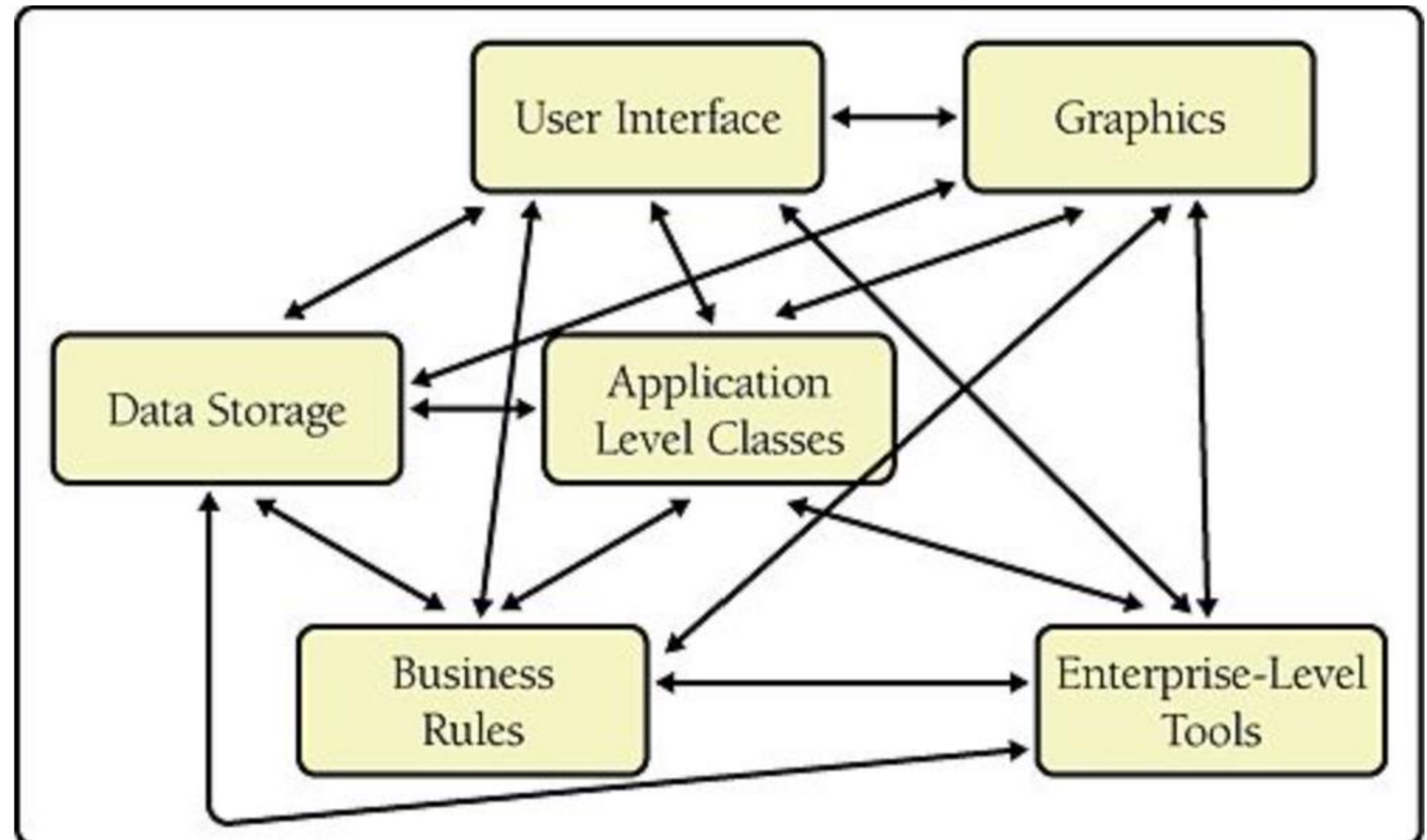
How many different parts of the system does a developer need to understand at least a little bit to change something in the graphics subsystem?

What's wrong with the following communication pattern?



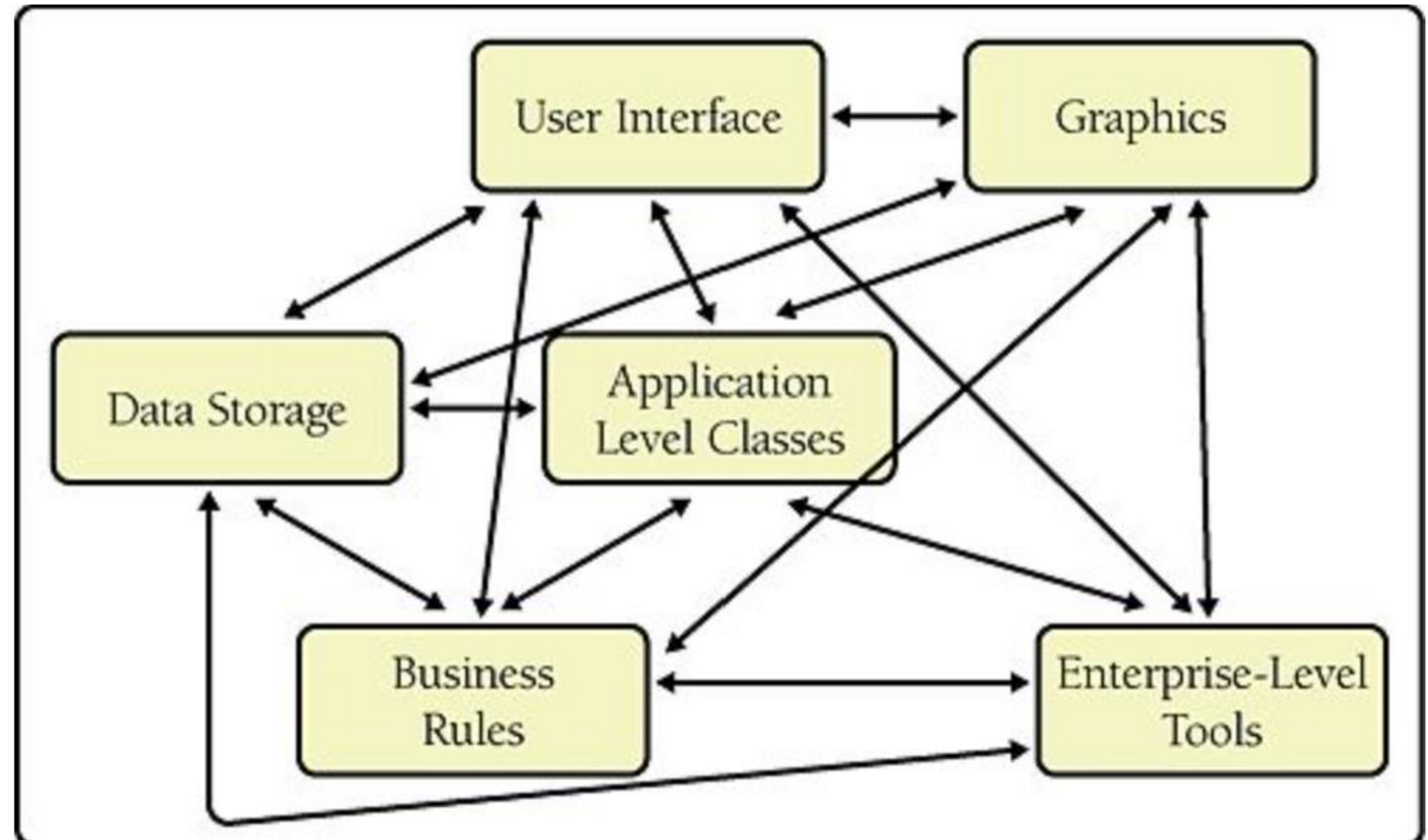
What happens when you try to use the business rules in another system?

What's wrong with the following communication pattern?



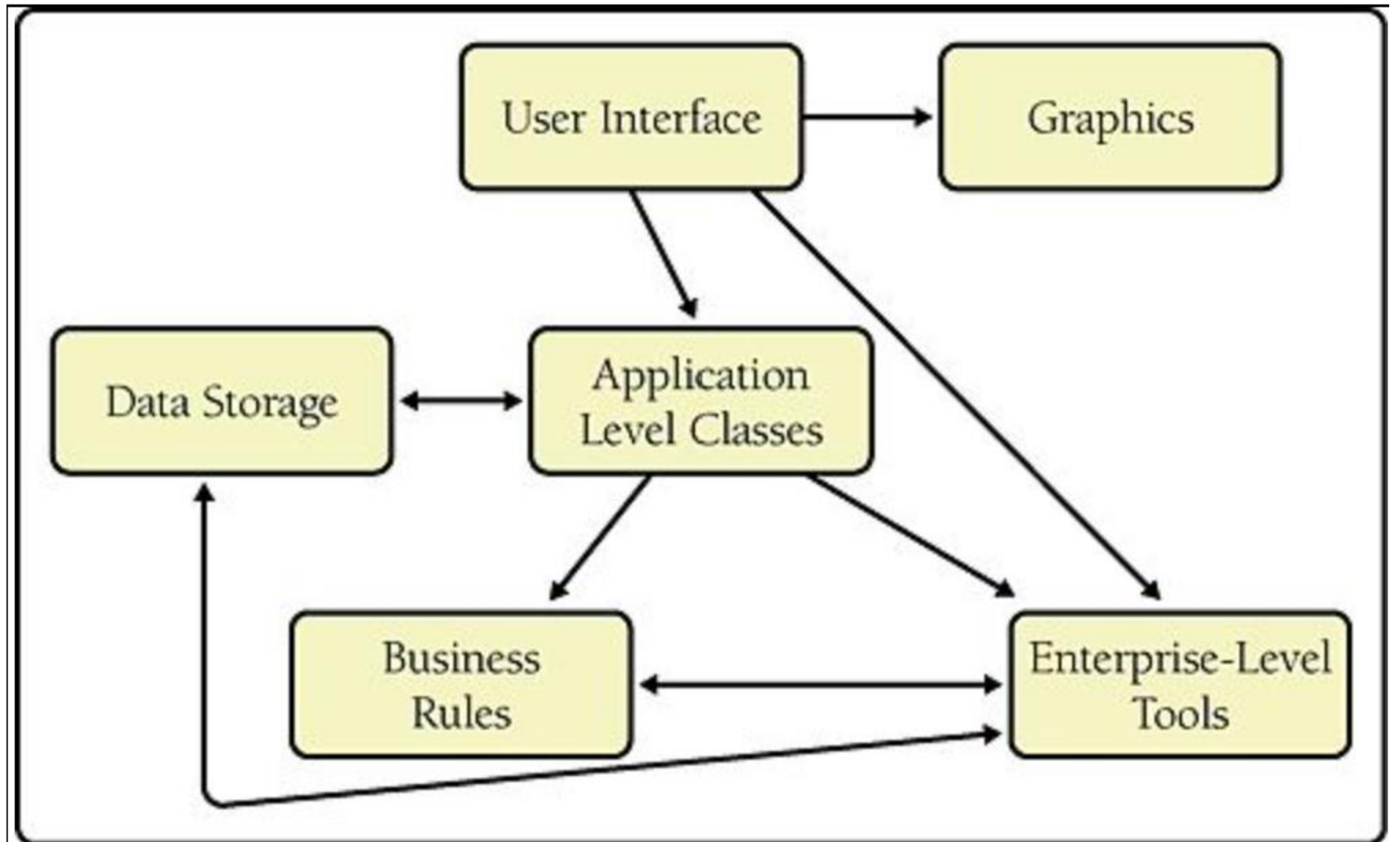
What happens when you want to put a new user interface on the system, perhaps a command-line UI for test purposes?

What's wrong with the following communication pattern?



What happens when you want to put data storage on a remote machine?

Fewer communication rules simplify interactions significantly



Farmville

Collision Avoidance Logic

Moving Objects

Fixed objects

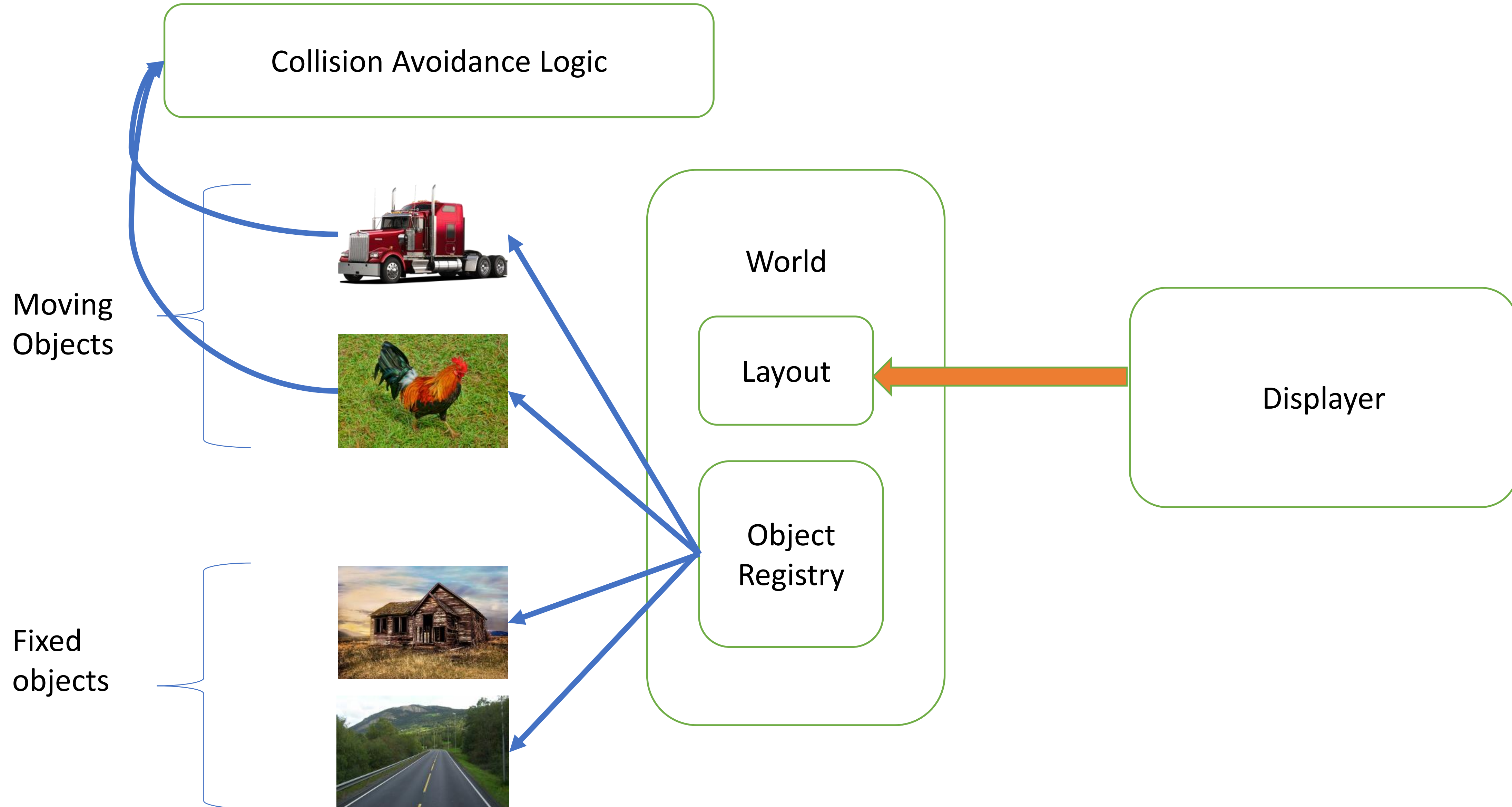
World

Layout

Object Registry

Displayer

Farmville



Hints for better subsystem design

- If not sure, err on the side of simplicity. Add more connections later.
- Three levels of relationship:
  - Simplest: One subsystem calls routines in another
  - More involved: One subsystem uses classes from another
  - Most involved: Classes in one subsystem inherit from classes in another
- Aim for acyclic dependencies

Level 3: Division into classes

- Identify all classes in each subsystem
- E.g. Database-interface subsystem can have
  - Data access classes
  - Persistence framework classes
  - Database metadata
- Identify class interfaces

Level 4: Division into Routines

- Interfaces are concerned with public routines
- Here we focus on the private helper routines
- Few functions may be very simple – getters, setters, constructors
- Some require more thought – interaction with other classes, calling multiple private routines, complicated algorithm
- Going through this process results in a better understanding of the class interfaces

Designing classes

- Identify the classes and their attributes
- Determine how other classes will interact with each class
- Determine how each class will interact with other classes
- Define the public interface of each class

Farmville classes: World

- Stores the layout of the world
- Manages the collection of all objects in a registry

Farmville classes: World

- Stores the layout of the world
- class Layout
  - manages pixels in an std::vector<std::vector<Space>> spaces
  - height is the size of spaces, width is the size of elements of spaces
  - class Space: contains a character, color, and a (possibly null) pointer to the object in it
- Manages the collection of all objects in a registry
- class ObjectRegistry
  - std::vector<std::shared_ptr<_Object>> objects

Farmville classes: World

- Contains variables for facilitating synchronization between the display thread and the object threads
- Manages the time elapsed since the beginning

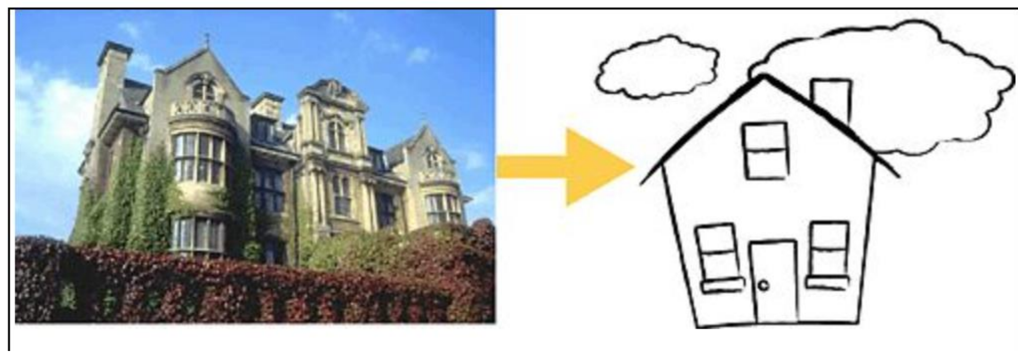
Farmville classes: World

- Contains variables for facilitating synchronization between the display thread and the object threads
  - std::shared_mutex display_mutex
  - std::condition_variable_any display_cv
- Manages the time elapsed since the beginning
  - uint64_t num_ticks = 0
  - uint32_t timescale_ms = 1000

Farmville classes: Displayer

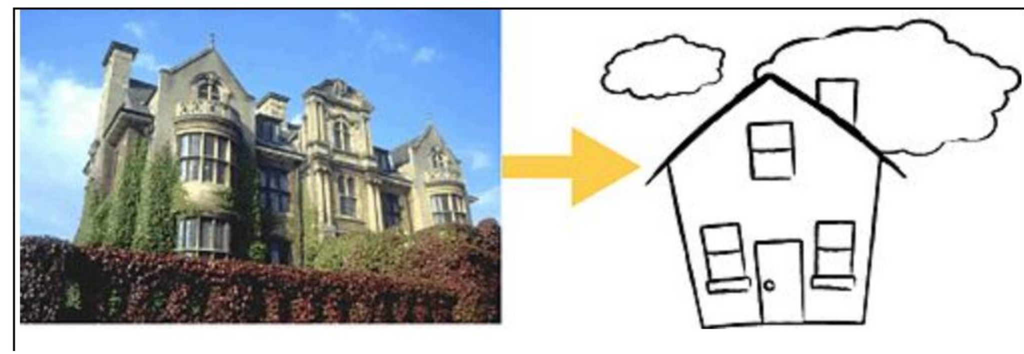
- Manage a thread that redisplays periodically
- Contains functions void display() and void display_loop()
- Delegates the actual displaying to a DisplayEngine object
- Maintains a shared pointer to the World

Form consistent Abstractions



- Abstraction is handling different details at different levels
- It allows you to ignore irrelevant details
- Understand the problem and form higher-level abstractions

Form consistent Abstractions



- Abstraction is handling different details at different levels
- It allows you to ignore irrelevant details
- Understand the problem and form higher-level abstractions
- Examples of abstractions
  - (Inheritance) Base classes contain common attributes of a set of derived classes
  - Class interface allows you to focus on the behavior of the class without worrying about the internal details
  - Thinking in terms of subsystems helps you focus on the overall behavior without worrying about the classes they are composed of
- Encapsulation enforces abstraction by disallowing access to lower-level details

Farmville: Using inheritance for objects

- The world only has access to a registry of _Object (base class)
- class _Object
  - Contains form of the object (a vector of strings)
  - Stores color
  - Stores position of the object (x-component and y-component)

Farmville: Using inheritance for objects

- The world only has access to a registry of _Object (base class)
- class _Object
  - Contains form of the object (a vector of strings)
  - Stores color
  - Stores position of the object (x-component and y-component)
- classes FixedObject and MovingObject inherit from it
- MovingObject supports relative positioning
  - Instead of the initial position, the constructor takes a reference object and the position relative to it

Farmville: Helper classes

- enum Color defines all supported colors
- enum Position defines all relative positions (north, north west, …)
- Commonly used object forms are defined in a separate file for reuse
    - barn_prototype
    - nest_prototype
    - chicken_prototype

```
std::vector<std::string> barn_prototype = {
        "          ",
        "   __ ^   ",
        "  /  /  \\ ",
        " |  | _  |",
        " |  |[ ] |"
};
```

Farmville: Creating world and objects

- Creating world and initializing display

```
World world(50, 50);
Displayer displayer(world);
```

Farmville: Creating world and objects

- Creating world and initializing display

```
World world(50, 50);
Displayer displayer(world);
```

- Adding static objects – barn and nests

```
std::shared_ptr<FixedObject> barn
        = std::make_shared<FixedObject>(
                barn_prototype,
                Color::BLACK,
                10, 10);
world.add_object(barn);
```

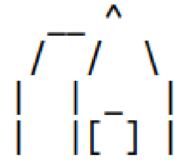
```
std::shared_ptr<FixedObject> nest1
        = std::make_shared<FixedObject>(
                nest_prototype,
                Color::BROWN,
                10, 25);
std::shared_ptr<FixedObject> nest2
        = std::make_shared<FixedObject>(
                nest_prototype,
                Color::BROWN,
                25, 25);
world.add_object(nest1);
world.add_object(nest2);
```

Farmville: Creating world and objects

- Creating moving objects – chickens

```
std::shared_ptr<MovingObject> chicken1
        = std::make_shared<MovingObject>(
                chicken_prototype,
                Color::BLUE,
                barn, Position::SOUTH_EAST,
                world);
std::shared_ptr<MovingObject> chicken2
        = std::make_shared<MovingObject>(
                chicken_prototype,
                Color::RED,
                barn, Position::SOUTH_WEST,
                world);
world.add_object(chicken1);
world.add_object(chicken2);
```

How does it look?



```
  0>            0>
^( )          ^( )
 = =           = =



  \     /        \     /
   -----          -----
```

What other forms of abstraction can we implement in Farmville?

What other forms of abstraction can we implement in Farmville?

- Incorporate movement patterns
  - Linear movement, piecewise movement, small random deviations

What other forms of abstraction can we implement in Farmville?

- Incorporate movement patterns
  - Linear movement, piecewise movement, small random deviations
- Implement path finding
  - From a given reference to another

What other forms of abstraction can we implement in Farmville?

- Incorporate movement patterns
  - Linear movement, piecewise movement, small random deviations
- Implement path finding
  - From a given reference to another
- Different objects have different behavior
  - For example, a chicken moves from nest to nest until it can find space to lay an egg
  - A nest changes its form upon receiving an egg
  - Specialize FixedObject and MovementObject classes to represent chickens, nests, truck, farmer etc.

Level 5: Internal routine design (implementation details)

- **How to guarantee that draw and redisplay synchronize properly and efficiently?** Reader-writer pattern
- Each object thread that redraws obtains a shared lock to display_mutex
- The redisplay thread obtains a unique lock to display_mutex

Level 5: Internal routine design (implementation details)

- **How to guarantee that draw and redisplay synchronize properly and efficiently?** Reader-writer pattern
- Each object thread that redraws obtains a shared lock to display_mutex
- The redisplay thread obtains a unique lock to display_mutex
- **When does redraw trigger?** When redisplay is done. The redisplay thread notifies all on display_cv
- **What is the condition for the drawing threads to wake up?** The redisplay thread increments num_ticks after redisplay, the drawing threads compare it with the last tick value they saw before wait
- The speed of an object is the inverse of the tick difference it redraws on!

Level 5: Internal routine design (implementation details)

- **How to avoid object collisions?** Implement object vision
- When an object detects a moving object nearby, it can either probe the object to learn its pathing or track its movement over time

Level 5: Internal routine design (implementation details)

- **How to avoid object collisions?** Implement object vision
- When an object detects a moving object nearby, it can either probe the object to learn its pathing or track its movement over time
- **To avoid deadlocks**, we can implement collision avoidance techniques such as moving in the opposite or orthogonal direction
- **To avoid livelocks**, we can implement random small halts and redirections

There is a lot more to design!

- We can’t explore all the topics here
- Some other topics include how to design the coroutines efficiently, how to enforce encapsulation, writing pseudocode, top-down and bottom-down design approaches and so on
- Are we done with Farmville design?

  No, far from it! Implementing the remaining features will expose a lot of incomplete ends, and also force us to redesign some components

Aside: Displaying using Emacs Lisp

- I implemented a class ElispEngine for the redisplay
- It makes use of emacs font colors
- Writes a file that can be executed with the emacs-lisp interpreter
- One can similarly write multiple different engines
  - Standard shell output
  - Using OpenGL 2D/3D graphics

```
class ElispEngine {
public:
    std::string output_file = "farmville.el";

    void write_header(std::ofstream& fout);
    void write(std::ofstream& fout,
               char ch, Color color = Color::BLACK);
    void write_newline(std::ofstream& fout);
};
```

Regular testing

- Test after adding each feature
- For example, I tested after I added static objects
- Then I tested after I added positioning a moving object in reference to a static object
- After I add collision avoidance, I will add two chickens that are approaching each other and observe what happens
- Write different test cases in different files with their own main function

Summary

- Software design is vital to writing good code
- If you find yourself getting stuck whenever starting a coding assignment, think through the problem and design a solution. Incrementally improve the design through preliminary versions of the code
- With Farmville, we focused on making the process of adding more objects and encoding their behavior smooth
- This process is slow in the beginning, but accelerates development after the groundwork has been laid
- Makes it much easier to reason about correctness and maintain code over time
- Remember – practice a lot if you want to get better!