B.U.B.O.L.O

Clone Productions

BU MET CS 673: Software Engineering | Spring 2014

Team Objectives

- Participate in a fun and exciting project
- Learn as many aspects of the Software Engineering process as possible: design & architecture, implementation, QA, etc
- Reinforce what we've learned in other courses (AI, Computer Graphics, Computer Networks, Design Patterns, Advanced Java, etc)
- Work in technical areas in which we had little exposure
- Games can involve a large number of subsystems, providing many learning opportunities

Team Objectives: Game Selection

- 8 game ideas were submitted to the wiki
- The game that we selected would represent our “customer”, so we wanted to pick an idea based on an existing game
- This would allow us to focus on the engineering rather than game design

Team Objectives: Bolo

- Bolo, a networked real-time game that was originally created in 1987, was selected unanimously:
  - Multiple team members had played the original version
  - It had many features (subsystems) that would allow for parallel development: Graphics, Sound, Networking, User Input, AI, GUI, etc
  - The project would scale well to match our skills
    - We'd still have a fine game if we had needed to cut the networking component, for example (but fortunately, we didn't)

Team Objectives: Bolo

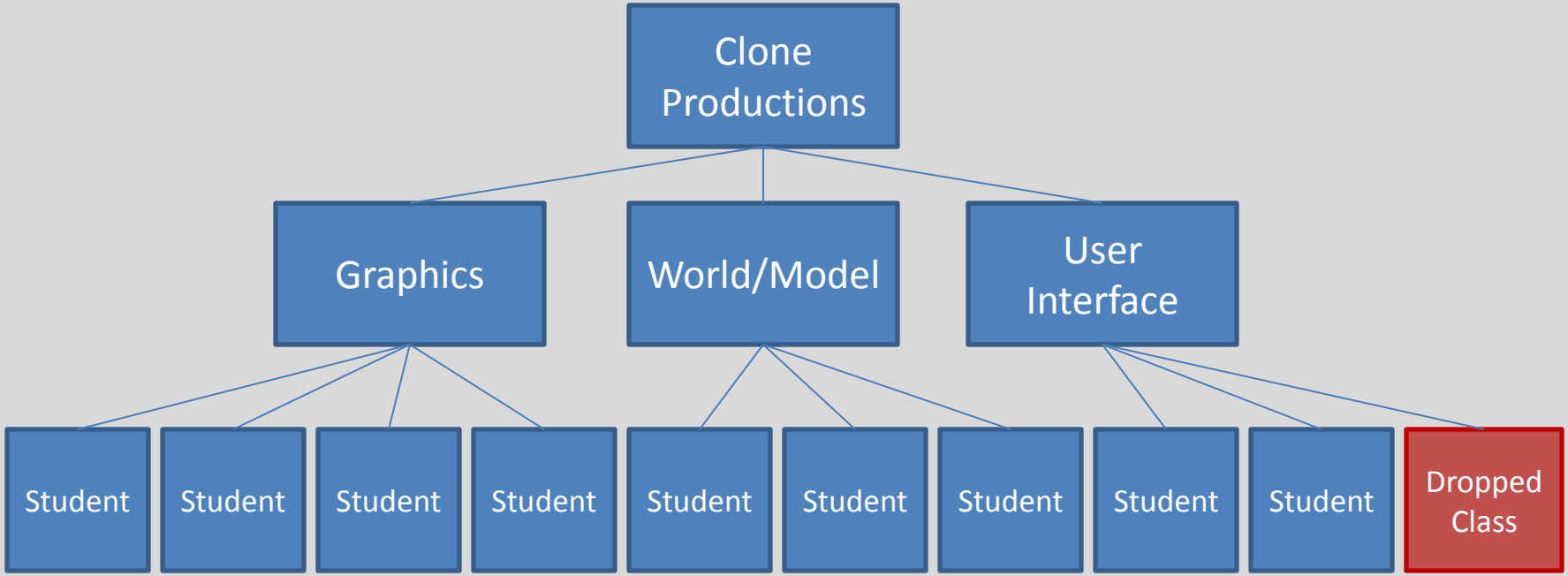
Screenshot of original Bolo

Team Organization & Skills

- Wide range of skills:
  - Multiple students had taken 10-20+ undergrad and graduate Computer Science classes
  - Others had taken few or no technical classes
  - Day jobs: software developers, project managers, full time students, …
  - Common denominator from an implementation-perspective: Java; nearly everyone had used it before
  - Multiple students had worked on games in their free time, but no one had released one

Team Organization & Skills

Initial organization: 3 Teams



Team Organization & Skills

- After Sprint 1, we voted to eliminate the subteam structure in favor of a task-focused structure
- This greatly increased flexibility, which was important since we had a small team and a large project
- Flat structure:
  - We're all colleagues
  - Project Manager was a facilitator
  - Team members volunteered for tasks

Development Process

- Agile process
- Requirements driven by User Stories, which were created early in the process
- Training sessions for Git/GitHub, Environment setup, Screen vs World coordinates, etc
- Environment setup guide on wiki
- Design documentation on wiki
- Extensive design & implementation discussions during Saturday meetings
- Javadoc pages created using Doxygen and available on GitHub organization site

Development Process

- We met in person on every Saturday throughout the semester
- We communicated through Google Groups and GitHub
- Source Control, wiki & Issue Tracker: GitHub
- Documentation: Google Docs; final versions of documentation exported to PDF and contributed to repository

Development Process

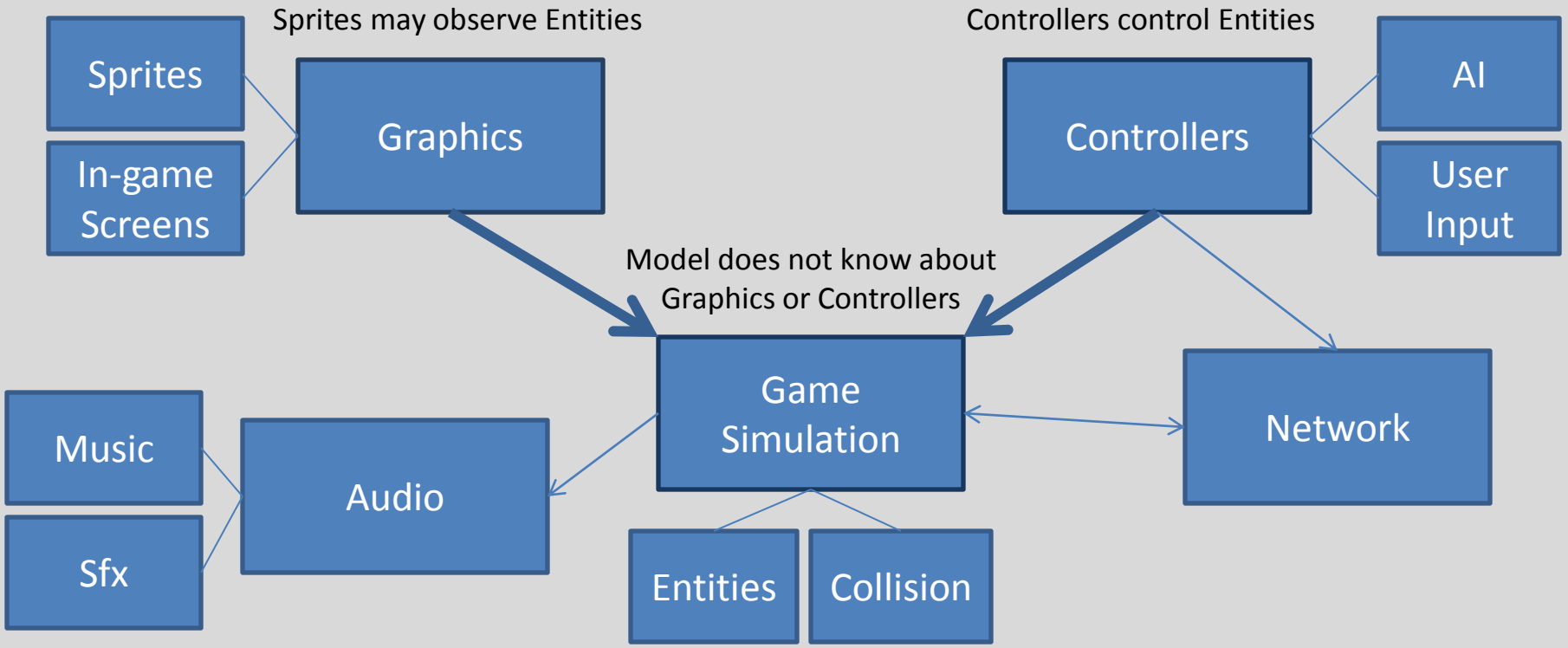
- Multi-tier review process, defined in SQAP, ensured that our project never encountered critical bugs in the production branch
- All code was reviewed by another developer before being merged into production:
  1. Javadoc comments required for all non-private methods & members
  2. Extensive compiler warnings
  3. FindBugs static code analysis tool
  4. Unit tests required for all non-private methods (364 unit tests at last count; 62.4% statement coverage)
  5. Integration tests required for larger features (13 integration test applications created)
  6. Enforcement of code standards, to ease readability & maintainability

Development Process: Changes

- Subteams eliminated
- Initially didn't utilize the Issue Tracker fully:
  - In Sprint 1, some of the subteams used Google Doc spreadsheets to sign up for tasks
  - We now use the issue tracker extensively, and it greatly benefits us
- Initially had a requirement that all pull requests were reviewed by two reviewers, the subteam lead and the QA manager, but this rule was eliminated when we eliminate the subteams
  - All code continues to be reviewed by an independent reviewer

Design Patterns

Model-View-Controller: High-Level Design



Design Patterns

Factory Method:

- World.addEntity: enables factory to be passed in with a request to create a new Entity, which enables the caller to specify the controllers that will control the Entity
- Sprite system uses factory method to map Entity types to Sprite types: sprite factory is abstract, and subclasses override the create method to return the correct sprite type

Design Patterns

Singleton:

- Networking

```
Network net = NetworkSystem.getInstance();
net.connect(ipAddress);
```

- Graphics class, which represents the graphics system
  - Despite implementing the singleton pattern, getInstance() has package-private visibility, rather than public
- Sprite creation & storage system
- And a few more

Design Patterns

Command Pattern: Networking

- Instead of sending data, we send commands through the network: we send behaviors
- Objects sent through the network must implement the NetworkCommand interface, and override `execute(World)`
- Works well, since TCP is guaranteed to deliver packets in the order they were sent
- All entities have unique IDs, which allows the command to get access to any entities it needs:

```
Entity entity = world.getEntity(id);
```

Design Patterns

Observer Pattern: Networking

- Other systems can become NetworkObservers, which allows them to register to receive events from the network
- Events include: onConnect; onClientConnected; onClientDisconnected; onGameStart; onMessageReceived
- Implemented by the multiplayer game lobby screen

Design Patterns

Strategy Pattern: Controllers->Entities

- Controllers represent the strategy of Entities
- An entity's controllers can be added or removed as needed at runtime

Template Method

- Screen: defines a final update method, which performs a number of steps
- Inheriting classes can override onUpdate hook method, which is called once per call to update; the default behavior is noop
- Similarly, the Entity.dispose final method has an onDispose non-final method that can be overridden by children

What was Achieved

^ Bolo

Bubolo >

What was Achieved

- All high-level objectives were reached
  - Real-time networked multiplayer game
  - Includes: Movable tank, collision detection, graphics, audio (music & sound effects), simple AI, network play with n players, more
  - Tested on: Windows, Mac OS X, Ubuntu Linux
- User stories intentionally included more than we expected to be able to complete in one semester, and had multiple stretch goals
  - This allowed us to scale the project if we accomplished more than we expected

What was Achieved

- Metrics:
  - Game: ~15,000 lines of code (carriage returns in java files); lines of code is a notoriously bad metric, but it can be useful to show scope
  - Tests: ~11,000 lines of code
  - Game source files: ~149
  - Test source files: ~112
  - Other: ~121, excluding generated Javadoc files
  - Tests: 364 unit test; 13 integration tests + 2 sprint tests
  - 392 issues opened; 360 closed

Top 3 Lessons Learned

1. Good architecture matters!
   - Our architecture greatly eased the implementation process, and enabled highly parallel development

2. Overuse of inheritance can lead to a brittle structure, and reduces flexibility
   - A wide and deep hierarchy in one of the subsystems led to refactoring and many discussions, and made implementing certain features difficult

3. Team dynamics is a non-trivial component of the software engineering process
   - Different opinions are inevitable and having the right experience and tools is critical to moving forward

Demo & Questions