Battleship overview

- What are the use cases?
  - How does customer use the program?
  - What are scenarios as the game develops?
  - What parts of the "standard version" are good/bad?
  - What options might we want to have?

- How will we design the program?
  - Brainstorm classes
  - Develop and test
  - Rethink design and use cases
  - Develop and test
  - …

Battleship classes, Freecell classes

- What are the classes in the program? Behaviors?
  - Look for objects, how do they act? Nouns? Verbs
- What about a Ship class? Behaviors/Responsibilities?
  - State? Mutable?
  - Comparison? Other games?
  - Is there any behavior?
- What about CardPile classes, similarities? Differences?
  - FreeCell, AcePile, DrawPile, …
  - Other card games?

Inheritance (language independent)

- First view: exploit common interfaces in programming
  - iterator, C++ function objects
    - Iterators in STL/C++ share interface by convention/templates
  - Implementation varies while interface stays the same
- Second view: share code, factor code into parent class
  - Code in parent class shared by subclasses
  - Subclasses can *override* inherited method
    - Can subclasses override and call?
- Polymorphism/late(runtime) binding (compare: static)
  - Actual function called determined when program runs, not when program is compiled

Inheritance guidelines in C++

- Inherit from Abstract Base Classes (ABC)
  - one pure virtual function needed (=0)
    - Subclasses must implement, or they're abstract too
  - must have virtual destructor implemented
    - can have *pure* virtual destructor with an implementation, but this is special case, not normally needed [force ABC]

- Avoid protected data, but sometimes this isn't possible
  - data is private, subclasses have it, can't access it
  - keep protected data to a minimum
- Single inheritance, assume most functions are virtual
  - multiple inheritance ok when using ABC, problem with data in super classes
  - virtual: some overhead, but open/closed principle intact

Inheritance Heuristics

- A base/parent class is an interface
  - Subclasses implement the interface
    - Behavior changes in subclasses, but there's commonality
  - The base/parent class can supply some default behavior
    - Derived classes can use, override, both
  - The base/parent class can have state
    - Protected: inherited and directly accessible
    - Private: inherited but not accessible directly
  - Abstract base classes are a good thing
- Push common behavior high up in an inheritance hierarchy
- If the subclasses aren't used polymorphically (e.g., through a pointer to the base class) then the inheritance hierarchy is probably flawed

Inheritance Heuristics in C++

- One pure virtual (aka abstract) function makes a class abstract
  - Cannot be instantiated, but can be constructed (why?)
  - Default in C++ is non-virtual or *monomorphic*
    - Unreasonable emphasis on efficiency, sacrifices generality
    - If you think subclassing will occur, all methods are virtual
  - Must have virtual destructor, the base class destructor (and constructor) will be called

- We use public inheritance, models *is-a* relationship
  - Private inheritance means is-implemented-in-terms-of
    - Implementation technique, not design technique
    - Derived class methods call base-class methods, but no "usable-as-a" via polymorphism
    - Access to protected methods, and can redefine virtual funcs

Inheritance and Layering/Aggregation

- Layering (or aggregation) means "uses via instance variable"
  - Use layering/attributes if differences aren't behavioral
  - Use inheritance when differences are behavioral

- Consider Student class: name, age, gender, sleeping habits
  - Which are attributes, which might be virtual methods

- Lots of classes can lead to lots of problems
  - It's hard to manage lots of classes in your head
  - Tools help, use speedbar in emacs, other class browsers in IDEs or in comments (e.g., javadoc)
- Inheritance hierarchies cannot be too deep (understandable?)

Inheritance guidelines (from Riel)

- Beware derived classes with only one instance/object
  - For the CarMaker class is GeneralMotors a subclass or an object?

- Beware derived classes that override behavior with a no-op
  - Mammal class from which platypus derives, live-birth?

- Too much subclassing? Base class House
  - Derived: ElectricallyCooledHouse, SolarHeatedHouse?

- What to do with a list of fruit that must support apple-coring?
  - Fruit list is polymorphic (in theory), not everything corable

Spreadsheet: Model, View, Controller

- **Model, View, Controller is MVC**
  - ➢ **Model stores and updates state of application**
    - **Example: calculator, what's the state of a GUI-calculator?**
  - ➢ **When model changes it notifies its views appropriately**
    - **Example: pressing a button on calculator, what happens?**
  - ➢ **The controller interprets commands, forwards them appropriately to model (usually not to view)**
    - **Example: code for calculator that reacts to button presses**
    - **Controller isn't always a separate class, often part of GUI-based view in M/VC**

How do Model/View communicate?

- Model has-a view (or more than one)
  - Can call view methods
  - Can pass itself or its fields/info to view

- View can call back on model passed (e.g., by model itself)
  - Model passes `this`, view accepts Model as parameter
  - Possible for controller/other class to pass model

- Controller contains both model and view (for example)
  - Constructs MV relationship
  - Possible for controller to be part of view (e.g., GUI)

Controller in MVC

- Loop until game over, where is code for board display?

```
while (true) {
    getMove(m,player);
    if (ttt.makeMove(m)){
        if (ttt.gameOver()){
            break;
        }
        player = (player == 'X' ? 'O' : 'X');
    }
    else {
        cout << "bad move " << m << endl;
    }
}
```

GUI controller

- Typically no loop, GUI events drive the system
  - Wire events to event handlers (part of controller)
  - What about model/view game over coordination?

```
connect(mouseClick, moveGenerator);  // metacode
void GUI::moveGenerator(MouseClick m)
{
    controller->process(moveFromMouse(m));
}
void Controller::process(const TTTMove& m)
{
    if (! myModel->makeMove(m)){
        myView->showBadMove(m);
    }
}
```

Designing classes in general

- Highly cohesive
  - Each class does one thing
  - Interface is minimally complete, avoid kitchen sink
    - What if client/user might want to hammer with an awl?

- Loose coupling (and minimize coupling)
  - Classes depend on each other minimally
  - Changes in one don't engender changes in another
  - Subclasses are tightly coupled, aggregates are not
    - Prefer Has-a to Is-a

- Test classes independently
  - Unit testing means just that, and every class should have a unit test suite

Tell/ask and the Law of Demeter

- "Don't talk to strangers"
  - Call methods in this class, parameters, fields, for created local variables, for values returned by class methods
  - No good, why? `fromPile.topCard().getSuit()`

From David.E.Smyth@jpl.nasa.gov Mon May 26 17:33:30 1997
>From: "David E. Smyth"
>To: lieber@ccs.neu.edu >Subject: Law of Demeter >
>I have been using LoD pervasively since about 1990, and it has taken
>firm hold in many areas of the Jet Propulsion Laboratory. Major systems
>which have used LoD extensively include the Telemetry Delivery System (a
>real-time database begun in 1990), the Flight System Testbed, and Mars
>Pathfinder flight software (both begun in 1993). We are going to use LoD
>as a foundational software engineering principle for the X2000 Europa
>orbiter mission. I also used it within a couple of commercial systems
>for Siemens in 91-93, including a Lotus Notes like system, and a email
>system.

More heuristics (some from Riel)

- Users depend on a class's interface, but a class shouldn't depend on its users

- Be suspicious of "God"-classes, e.g., Driver, Manager, System
  - Watch out for classes supporting method subsets

- Beware of classes with lots of get/set methods

- Support Model/View distinction
  - The model shouldn't depend on the view, but should support multiple views

- If a class contains an object it should directly use the object by sending it messages

Working as part of a group

see McCarthy, *Dynamics of Software Development*

- establish a shared vision
  - what was/is Freecell? what can we add?
  - harmonious sense of purpose
- develop a creative environment
  - the more ideas the better, ideas are infectious
  - don't flip the BOZO bit
- scout the future
  - what's coming, what's the next project
  - what new technologies will affect this project

Scheduling/Slipping

- McCarthy page 50, Group Psyche, TEAM=SOFTWARE
  - anything you need to know about a team can be discovered by examining the software and vice versa
  - leadership is interpersonal choreography
  - greatness results from ministrations to group psyche which is an “abstract average of individual psyches”
  - mediocrity results from neglect of group psyche
- Slipping a schedule has no moral dimension (pp 124-145)
  - no failure, no blame, inevitable consequence of complexity
  - don’t hide from problems
  - build from the slip, don’t destroy
  - hit the next milestone, even if redefined (“vegetate”)

Towards being a hacker

- See the hacker-faq (cps 108 web page)
  - Hackers solve problems and build things, and they believe in freedom and voluntary mutual help. To be accepted as a hacker, you have to behave as though you have this kind of attitude yourself. And to behave as though you have the attitude, you have to really believe the attitude.

- The world is full of fascinating problems
  - no one should have to solve the same problem twice
  - boredom and drudgery are evil
  - freedom is good
  - attitude is no substitute for competence

You may not work to get reputation, but the reputation is a real payment with consequences if you do the job well.

Aside: ethics of software

- What is intellectual property, why is it important?
  - what about FSF, GPL, copy-left, open source, …
  - what about money
  - what about monopolies

- What does it mean to act ethically and responsibly?
  - What is the Unix philosophy? What about protection? What about copying? What about stealing? What about borrowing?
  - No harm, no foul? Is this a legitimate philosophy?

- The future belongs to software developers/entrepreneurs
  - what can we do to ensure the world's a good place?