

CS249 Final Exam: Closed Book, 180 Minutes

Instructor: David Cheriton, Tuesday December 11, 2005

December 3, 2007

This is a CLOSED-BOOK exam. You are expected to work on your own to complete this exam, not using other people or any materials for your assistance.

- Please answer all questions; read each question carefully. You waste time by writing more than required by the question, and you lose points by answering the wrong question.
- Please give precise, specific answers that demonstrate the depth of your knowledge of the area being examined by the question, rather than vague and general comments. Point form answers are acceptable. You can use sketch code that is not necessarily complete AS LONG AS IT CLEARLY INDICATES you understand the approach or technique we are after.
- Please put your name and student number on each exam booklet you use and sign the honor code statement.

1. 30 Points

- (a) Paten Gaelic, a smarty junior programmer who you hired, claims it is absurd to insist that accessors never throw exceptions in real software because there are lots of things that can go wrong in reality, from "illegal index" to "entry not found" to memory corruption to network errors (in the case of a distributed object system) to storage errors with persistent objects. "Prove" that you never have to throw exceptions in any real world setting if you follow the CS249 regime.
- (b) Paten also claims that transactional semantics for mutators is an academic fantasy that is infeasible in lots of cases. For example, if you tell a document to print itself, an airplane to take off or a cash dispenser to give out money, you cannot undo the action by just rolling back the state. Argue why you can always implement transactional semantics in a mutator if you follow the CS249 regime, or argue that you can't.

2. 30 Points Paten regards Cheriton's "holy trinity" of entity, value and named description types as an unnecessary complication.

- (a) He proposes unifying the treatment of all types of instances, namely as things one could point to, i.e. basically entity-like. For example, a "value" attribute would have a pointer to its value, the same as an entity attribute would have. All instances would be passed by reference, so there would be no more overhead passing around large values than small, seemingly obviating the need for named description types. Describe the key problem with this approach.
- (b) He then leaps to the alternative of treating all types as value types including our so-called entity types. You could pass objects by C++ const reference for efficiency, as we do now for some values. You could define equality on these "entity" objects based on their name, so you could identify two copies of an entity as being the same entity. Describe the key problem with this approach.

3. 30 Points Considering events and notification, Paten also does not "believe" in having all events being handled within Cheriton's restrictive "notification" model. For instance, the system can have events corresponding to device failure, packet arrival, mode change, counter overflow, etc.

- (a) Describe in general why it is feasible to fit every software event into the notification model and illustrate in code for an example type of event, ideally one that is not obvious how it might fit.
- (b) Paten does not "get" how to put all this CS249 stuff together, like how do you actually structure an object with interfaces, etc. Draw a figure illustrating the configuration, reporting and reactor elements for implementing an `Airplane`, labeling it according to what is readable and writable by each component, and indicating the data flow. Further, illustrate how a `Missile` object should be structured as a component of `Airplane`.
4. **30 Points** Paten Gaelic writes the following "carefully nested try block" code to create a file, giving you the MTV wisecrack that Cheriton is just a useless academic who has never written "real" software with exceptions.

```

FileManager::create( FileName n ) {
    // create a file called n, as file descriptor and directory
    // entry.
    DirEntry * de = 0;
    FileDesc * fd = 0;
    try {
        de = newDirEntry(n);
        if( de ) try {
            fd = newFileDesc();
            if( fd ) try {
                de->fileDescIs(fd);
            } catch(...) {
                delete fd;
                delete de;
                throw FileCreationException;
            }
        } else {
            delete de;
            throw FileCreationException;
        }
    } catch(...) {
        throw;
    }
    else throw DirEntryFailed;
}
catch( DirEntryFailed& def ) {
    throw FileCreationException;
}
}

```

To put this smarty boy in his place,

- (a) Rewrite this function with as few try blocks and throw statements, etc. as possible, stating and justifying any assumptions you are making about the functions that are called.
- (b) Describe to Paten why your rewritten version is superior from a software engineering standpoint, including pointing out any problems that his code has.
5. **30 Points** Paten finds Cheriton's claim that the `Ptr` class template is basically the only form of smart pointer required absurd, given the infinite number of classes that could override `operator->` to provide "rich" semantics and all the different types of objects you might want to point to, e.g. legacy, distributed, persistent, etc.

- (a) Describe how this "one size fits all" `Ptr` approach can work, and why this is the best approach (or argue that it is not).
- (b) To Paten with his GC background, the reference-counted `Ptr` approach has some significant disadvantages to deal with, including 1) cycles, 2) error-proneness in the misuse of raw vs. smart pointers, 3) overhead for space of reference counts, increment/decrement processing, and large-scale synchronous deletions on clearing/destroying a single pointer. Describe why these are not significant disadvantages relative to garbage collection (or argue that they are.)

6. **30 points** Considering composition and collisions:

- (a) With the collision approach being a very general mechanism for dynamic discovery and maintenance of inter-object relationships, Paten Gaelic finds the collision approach "cool" and goes for it in a big way, i.e. too much of it. Describe quantitatively its key disadvantage and three approaches to minimizing this disadvantage.
- (b) Paten gripes that Cheriton's infatuation with composition and component-oriented design over inheritance means you have to explicitly write "wiring" code and explicit conversion to the component pointer, rather than just letting the compiler do all the work, i.e. implicit upcasts, combining virtual function tables, etc. Considering a library designed for the (alternative) inheritance-based composition with (say) `TimeableObj`, `DisplayableObj`, `CollidableObj`, etc. and an overridable callback functions, e.g.

```
class TimeableObj {
    Name name() const;
    void virtual timeout() = 0;
};
```

illustrate in code three key disadvantages of this inheritance-based approach.

The End, Happy Holidays!