EC327 Spring 2025 Final Exam Instructions

Please write your name on the back of this exam. This will remove unconscious bias during the grading of the exam as the graders will not see names while grading.

This is a closed book exam, absolutely no technology, books, or notes are allowed to be consulted while taking the exam. Violating this rule will result in a 0 grade for the exam.

When asked to write code, you will not be evaluated on exact syntax, rather just the core concepts/ideas.

The exam will be graded out of 100 points and consists of the following problems:

1. True or False, General Knowledge **(14 points)**
2. Multiple Choice, General Knowledge **(12 points)**
3. Templates (**12 points)**
4. Inheritance & Function Invocation Dispatch (**12 points)**
5. Standard Template Library **(20 points)**
6. Android Development (**20 points)**
7. Composition & Inheritance (**10 points)**

True or False - General Knowledge (14 points - 2 points each)

(Whether you respond with True or False, please explain your response in a sentence). You will get 1 point if your True/False response is correct and another if your sentence clearly provides an explanation.

1. Dependency injection eliminates the need for polymorphism in object-oriented design.

RUBRIC

False - Dependency injection only works if you are set up for polymorphism. The class that a collaborator is being injected into can only accept arbitrary implementations of that collaborator if it refers to all of them via a common interface.

2. Event-driven programs in Android typically follow a top-down control flow like in procedural C code.

RUBRIC

False - The Android framework is based on an asynchronous, event driven programming model where events are queued up and processed by handlers that have been registered to listen to them.

3. In object oriented C++ code, stack-allocated objects require manual deletion to avoid memory leaks

RUBRIC

False - Just as with non object oriented C programming,memory associated with variables and objects allocated on the stack is reclaimed once the stack frame is popped.

4. In Scrum, the purpose of Story Point planning is coming up with a detailed plan for implementing the feature.

RUBRIC

False - It is to estimate relative effort or complexity to help with team planning

5. In Android development, UI elements can be freely accessed and updated from any background thread without any additional synchronization.

RUBRIC

False - UI objects should only be accessed from the UI thread. Background threads should not touch UI elements directly. Android UI components are not thread safe and concurrent manipulation could lead to corrupted UI state.

6. In C++, dynamic dispatch requires that the function being dispatched is declared as virtual

RUBRIC

True - Without the virtual keyword, function dispatch will be static, based on the compile-time type rather than the runtime type.

7. In object-oriented programming, it is generally a good idea to allow child classes to directly access private fields of their parent class.

RUBRIC

False - Private fields are not accessible by child classes. Protected fields can be accessed, but heavy reliance on inherited fields breaks encapsulation.

Multiple Choice (12 points - 2 points each)

1. Which of the following is a benefit of dependency injection?
    a. Tightly couples components for faster access
    b. Makes code harder to test
    c. Allows behavior to be swapped out at runtime
    d. Alleviates the need for manual memory management

RUBRIC: C

2. In Kanban, which of the following best describes how work is assigned?
    a. The project manager assigns tasks at the start of the sprint
    b. Developers pull tasks from the backlog continuously
    c. Tasks are assigned based on component ownership
    d. Tasks are assigned roughly 6 weeks in advance after a lengthy planning session

RUBRIC: B

3. In Scrum development, what is a “story point”?
    a. The product requirements for a feature from a user’s perspective
    b. A presentation on the evolution of requirements for a feature.
    c. A detailed description of the implementation plan of a feature
    d. A very rough estimate of the complexity and implementation time for a feature

RUBRIC: D

4. In a continuous integration (CI) pipeline, which of the following are most appropriate times to trigger test suite execution?
    a. When code is merged into the main branch
    b. Once a week at midnight
    c. When a Pull Request is created

d. A and C

RUBRIC: D

5. In C++, what is the purpose of the override keyword?
    a. Allow multiple constructors for a class
    b. Indicate that a function overloads another function
    c. Indicate that a function overrides a virtual function from a base class
    d. Make a function non-virtual

RUBRIC: C

6. What is a key difference between std::map and std::unordered_map in C++?
    a. std::unordered_map guarantees O(log N) lookups
    b. std::map is backed by a hash table, std::unordered_map by a balanced tree
    c. std::map maintains values in sorted order, std::unordered_map does not
    d. std::map maintains keys in sorted order, std::unordered_map does not

RUBRIC: D

Problem 3 : Templatized Class (12 points)

A Stack is a very useful container data structure that supports Last In - First Out behavior. You can picture a stack of dishes where the dish placed on top of the stack most recently is the dish that will get picked off of the stack next. This data structure is even more powerful if it can be reused across all data types. In other words, we should not need to implement the Stack for integers and then re-implement it for strings.

Your stack should support

- **push -** adds an element to the top of the stack
- **pop -** removes and element from the top of the stack
- **top -** returns the element at the top of the stack without removing it
- **isEmpty -** returns **true** if the Stack contains no elements
- **contains** - takes an element of the generic type and returns true if an element exists in the stack that is **==** to the passed in element. It is OK to assume that all generic types used for this Stack override the **==** operator.

Your implementation should use the **std::vector** container from the **STL.**

Problem 3.1 Declaration & Implementation (9 points)

Provide the **stack.h** declaration & implementation for your generic Stack. Both can be in the **.h** file to avoid templatized compilation challenges.

RUBRIC:

```
template<typename T>
class SimpleStack {
private:
    std::vector<T> elements;

public:
    void push(const T& value);
    void pop();
    bool empty() const;
    bool contains(const T& value) const;
};

template<typename T>
void SimpleStack<T>::push(const T& value) {
    elements.push_back(value);
}

template<typename T>
void SimpleStack<T>::pop() {
    elements.pop_back();
}

template<typename T>
T& SimpleStack<T>::top() {
    elements.back();
}

template<typename T>
bool SimpleStack<T>::empty() const {
    return elements.empty();
}

template<typename T>
bool SimpleStack<T>::contains(const T& value) const {
    return std::find(elements.begin(), elements.end(), value) != elements.end();
}
```

- template<typename T> - above class declaration. It is ok if they forget to repeat this above every method - **3 points**

- Correct templating of the underlying **vector** container **- 2 points**
- **push** - correct signature and implementation - **1 point**
- **pop** - correct signature and implementation - **1 point**
- **isEmpty**- correct signature and implementation - **1 point**
- **contains** - correct signature and implementation. Can use std iterators or their own loop with a == check - **1 point**

Problem 3.2 Performance (3 points)

What is the big-O running times for you **push, pop,** and **contains** implementations?

RUBRIC

- **push -** constant time, O(1) - **1 point**
- **pop -** constant time, O(1) - **1 point**
- **contains -** linear time, O(N) - **1 point**

Problem 4: Inheritance & Function Invocation Dispatch (12 points)

You are building a notification system that can send different kinds of notifications: email notifications and SMS (text) notifications, to start.

There is a base class **NotificationSender** that defines a **send**() method.

Each specific type of notification will inherit from NotificationSender and implement its own version of send().

Problem 4.1: Abstract Class (3 points)

Let's start by implementing the base class - **NotificationSender.**

It should have a void **send** function that takes a single **std::string** argument with a default implementation that simply prints a message about the base sender sending the specific message.

It should be declared in a way that causes dispatching to route invocations to its descendant classes if those are present at runtime.

It should also have a virtual destructor.

Write your **notificationsender.h** here:

RUBRIC

```
#include <iostream>

class NotificationSender {
public:
    virtual void send(std::string message) {
        std::cout << "Base Notification Sender sending " << message <<
std::endl;
    }

    virtual ~NotificationSender() {};
```

```
};
```

- The **send** function should be declared **virtual** - **2 points**
- There should be a virtual destructor - **1 point**

Problem 4.2: Implement SMSSender (2 points)

Now provide the implementation of **SMSSender.** This class should inherit from **NotificationSender** and provide its own implementation of the **send** function. For the purposes of the exam, the implementation can simply print out a message about SMSSender sending the specific passed in message.

SMSSender does not need to take any constructor arguments.

For the purposes of the exam, the entire implementation can be provided in **smssender.h** below:

RUBRIC

```
#include "notificationsender.h"

class SMSSender : public NotificationSender {
public:
    void send(std::string message) override {
        std::cout << "SMS Sender sending " << message << std::endl;
    }
};
```

- Correct inheritance syntax : ": NotificationSender" (no points off for forgetting public keyword) - **1 point**
- Correct overriding of **send** with **override** keyword (no points off for incorrect placement of **override** keyword)

Problem 4.3: Implement EmailSender (3 points)

Now, let's move onto implementing an email sender. It is going to inherit from **NotificationSender** and be similar to the **SMSSender** that you just implemented.

However, it has a private pointer to a server address and initializes this pointer via a constructor argument. It is its responsibility to clean up this memory address when it is destroyed.

For the purposes of the exam, the implementation of **send** can simply print out a message about EmailSender sending the specific passed in message.

For the purposes of the exam, the entire implementation can be provided in **emailsender.h** below:

RUBRIC

```
class EmailSender : public NotificationSender {
private:
    std::string* emailServer;
public:
    EmailSender(std::string* emailServer) {
        this->emailServer = emailServer;
    };

    void send(std::string message) override {
        std::cout << "Email Sender " << message << std::endl;
    }

    ~EmailSender() {
        delete emailServer;
    }
};
```

- Constructor should correctly initialize emailServer - **1 point**
- **send** should be correctly overridden and implemented - **1 point**
- Destructor should be overridden and **delete** emailServer called - **1 point**

Problem 4.4: Invocation (4 points)

Answer the questions below with the following code snippet in context:

```
int main() {
    std::string* emailsServer = new std::string("localhost");

    NotificationSender* sender = new EmailSender(emailsServer);
    sender->send("hi there");

    delete sender;

    return 0;
}
```

- When the **main** function is executed, what is printed out?
- Explain how the **send** function invocation is routed.
- Explain what destructor is invoked when the **main** function completes and how that invocation is routed.

RUBRIC

- "Email Sender hi there" is printed - **1 point**
- The **send** function is invoked via virtual dispatch - its compile time type is **NotificationSender** but its runtime type is **EmailSender** and because the function is **virtual**, the runtime type is used
    - Use of the term **dispatch** - **1 point**
    - Use of the term **virtual** - **1 point**
- The destructor implementation of **EmailSender** is invoked because **NotificationSender** has a virtual destructor - **1 point**

Problem 5: Standard Template Library (20 points)

For this problem, you will be designing a class that keeps track of all users of a messaging application that you are building. The **User** class has been defined for you as:

```
#include <string>

class User {
private:
    std::string username;
    int ageInDays;
public:
    User() {};

    User(std::string username, int ageInDays) :
        username(username), ageInDays(ageInDays) {}

    std::string getUsername() const {
        return username;
    }

    int getAge() const {
        return ageInDays;
    }

    std::string display() const {
        return username + " " + std::to_string(ageInDays);
    }
};
```

The core tasks that your **UserManager** will need to support are:

- **bool registerNewUser(const User* user)**
    - asks the **UserManager** to keep track of a new **User**. This method should return **true** if this user has not yet been registered with the **UserManager** and now is; and return **false** if the **UserManager** has seen this **User** before and therefore did not update anything.
    - The implementation of this function should be O(log n).
- **const User* getUser(std::string username)**

    - asks the **UserManager** to retrieve the reference to the **User** with the given **username**. The function should return a null pointer if a user with the given **username** has not been registered with the **UserManager**.
    - The implementation of this function should be O(1).
- **const User* getYoungestUserOlderThan(int daysOld)**
    - asks the **UserManager** to retrieve the youngest known user who is older than **daysOld,** inclusive. If there is not a single user older than **daysOld,** inclusive, then this function should return a null pointer.
    - Please keep in mind that you may have multiple users who are the same age.
    - If multiple users are tied for this distinction (i.e. they are the youngest users over **daysOld** and are the same age), returning any one of them is fine.
    - The implementation of this function should be O(log N).

Problem 5.1: UserManager header (8 points)

In the space below, write your **usermanager.h** declaration:

RUBRIC

```
#include <map>
#include <string>
#include <unordered_map>

#include "user.h"

class UserManager {
private:
    std::unordered_map<std::string, const User*> usersByName;
    std::map<int, std::vector<const User*>> usersByAge;
public:
    bool registerNewUser(const User* user);
    const User* getUser(std::string username) const;
    const User* getYoungestUserOlderThan(int daysOld) const;
};
```

- **usersByName** should be an unordered map - there is no use case for traversing users in the order of their names **- 2 points**
- **usersByAge** should be an ordered map because we need to be able to find the youngest user older than X days - **2 points**
- Both of the maps should be **private -** 1 **point**

- The functions described in the instructions should have been properly converted into **public** declarations - **2 points**
- All interactions with **User** should involve **const User*** to avoid making copies of users. - **1 point**

Problem 5.2: UserManager implementation (6 points)

In the space below, write your **usermanager.cpp** implementation:

```
#include "usermanager.h"

bool UserManager::registerNewUser(const User* user) {
    if (usersByName.find(user->getUsername()) != usersByName.end()) {
        return false;
    }

    usersByName[user->getUsername()] = user;
    usersByAge[user->getAge()].push_back(user);

    return true;
}

const User* UserManager::getUser(std::string username) const {
    std::unordered_map<std::string, const User*>::const_iterator it =
usersByName.find(username);
    if (it != usersByName.end()) {
        return it->second;
    }
    return nullptr;
}

const User* UserManager::getYoungestUserOlderThan(int age) const {
    std::map<int, std::vector<const User*>>::const_iterator it =
usersByAge.lower_bound(age);

    for (; it != usersByAge.end(); ++it) {
        const std::vector<const User*>& usersAtThisAge = it->second;

        for (std::vector<const User*>::const_iterator userIt =
usersAtThisAge.begin();
             userIt != usersAtThisAge.end(); ++userIt)
        {
            return *userIt;
```

```
        }
    }

    return nullptr;
}
```

Can be very lenient on exact syntax here:

- **registerUser** - should maintain both of the maps needed for this implementation - **4 points (2 points each)**
- **getUser -** should involve no looping/iterations - **1 point**
- **getYoungestUserOlderThan** - should involve no non-short-circuited looping/iterators. Exit fast when we either find a user that meets the criteria or know that we don't have one - **1 point**

Problem 5.3: Implementation Choice - getUser (3 points)

The problem statement required that **getUser** be O(1). In a paragraph, explain how you accomplished this and how any underlying library or data structure that you used accomplishes this.

RUBRIC

- Up to 3 points among the following:
    - Mention hash map or hash table - **1 point**
    - Mention hash maps being backed by arrays - **1 point**
    - Mention good hash function behavior being dependent on avoiding hash collisions which requires a good, random hash function - **1 point**
    - Mention strategies for dealing with hash collisions - **1 point**

Problem 5.4: Implementation Choice - getYoungestUserOlderThan (3 points)

The problem statement required that **getYoungestUserOlderThan** be O(log N). In a paragraph, explain how you accomplished this and how any underlying library or data structure that you used accomplishes this.

RUBRIC

- Up to 3 points among the following:

- Mention STL map being backed by binary search trees- **1 point**
- Mention height of binary search trees being O (log N) - **1 point**
- Mention traversal of a binary search tree involving a comparison at every level - **1 point**
- Mention strategies for keeping BST's balanced, e.g. red-black trees - **1 point**

Problem 6: Android Development (20 points)

You are building a Shopping List application. The goal of the application is to let its users manage their grocery shopping lists, keep track of what they still need to purchase, and be notified of groceries that may be on sale via API calls.

You are starting with this code snippet from the **MainActivity:**

```java
public class MainActivity extends AppCompatActivity {
    private static final String PREFS_NAME = "ShoppingPrefs";
    private static final String KEY_ITEMS = "items";

    private EditText itemEditText;
    private Button addButton;
    private TextView shoppingListTextView;

    private List<String> shoppingItems = new ArrayList<>();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        itemEditText = findViewById(R.id.itemEditText);
        addButton = findViewById(R.id.addButton);
        shoppingListTextView = findViewById(R.id.shoppingListTextView);

        addButton.setOnClickListener(v -> {
            String item = itemEditText.getText().toString();
            shoppingItems.add(item);
            updateShoppingList();
        });

        loadItems();
    }


    private void updateShoppingList() {
        shoppingListTextView.setText(TextUtils.join("\n", shoppingItems));
    }

    private void loadItems() {
        SharedPreferences prefs = getSharedPreferences(PREFS_NAME, MODE_PRIVATE);
        String items = prefs.getString(KEY_ITEMS, "");
        shoppingItems = new ArrayList<>(Arrays.asList(items.split(",")));
        updateShoppingList();
    }

    private void saveItems() {
        SharedPreferences prefs = getSharedPreferences(PREFS_NAME, MODE_PRIVATE);
        prefs.edit().putString(KEY_ITEMS, TextUtils.join(",", shoppingItems)).apply();
    }
}
```

Problem 6.1: Code Analysis (6 points)

Problem 6.1.1: Event Registration (2 points)

What event is being registered in the **onCreate** method? What behavior is expected when this event is triggered?

RUBRIC

- **1 point** - The onClick listener is being registered for the **addButton**
- **1 point -** When the event is triggered, the value in the input box is read, added to the in-memory array stored on the **MainActivity**, and the text box listing all of the items is re-rendered with the updated list of items.

Problem 6.1.2: Non Blocking (2 points)

What core Android principle guides us not to perform any long-running or potentially blocking work in the code that is executed when the event is triggered?

RUBRIC

- **1 point** - The event handling will be executed by the Main Android thread (UI Thread)
- **1 point** - We don't want to block this thread because if we do, our UI will become unresponsive and we might trigger an ANR (Application Not Responding) error.

Problem 6.1.3: Quick Additions (2 points)

Support the user adds 50 items in quick sessions. Will the current code structure handle this interaction smoothly? Explain why or why not.

RUBRIC

- **1 point** - The current code will handle this interaction smoothly.
- **1 point -** All of the logic involves simple, lightweight, local operations so there won't be any issues.
- **NOTE:** if students bring up the concern that the entire list of items (up to 50 here) are repainted/re-rendered every time, this is a valid concern but at this volume won't be anywhere near enough to cause problems. They can still get full credit if they bring this concern up and say that this situation would not be handled smoothly.

Problem 6.2: Persistence (8 points)

Problem 6.2.1: As Implemented (2 points)

As implemented, what is the current persistence strategy for the application? In other words, where/when is data being saved/loaded?

RUBRIC

- **1 point -** items are saved into an in-memory array on the MainActivity.
- **1 point -** items are also loaded into that in-memory array from disk when the MainActivity is created but they are never written to disk.

Problem 6.2.2: Persistence Shortcomings (2 points)

What are the shortcomings of the current persistence strategy?

RUBRIC

- **1 point -** the in memory state will get lost if the MainActivity is ever destroyed. It is better to store in-memory state on the application, not on the activity.
- **1 point -** item state is never written to disk, only read from disk

Problem 6.2.3: Addressing Shortcomings (2 points)

If you identified any shortcomings above, address them now. You can address them by providing code snippets below or by explaining what you would do in words.

RUBRIC

- **1 point -** move the in-memory collection/list to the application object
- **1 point -** actually call **saveItems** when the application is being backgrounded, when the application is low on memory, or when the MainActivity is paused.

Problem 6.2.4: General Persistence Guidelines (2 points)

In general, what are some persistence guidelines and trade-offs for Android applications that we talked about in class? The code snippet above utilizes **SharedPreferences**. What other on-disk persistence options are you aware of? What are the trade-offs between the options you are familiar with?

RUBRIC

Up to 2 points among the following:

- **1 point -** use in-memory persistence for fast updates between activity navigation
- **1 point -** use on-disk persistence for storing data across app restarts.
- **1 point -** SharedPreferences used here are a lightweight map. Fine for small key-value like collections. File writing via a protocol like JSON - flexible and can handle large data. ROOM - Android's object-relational mapping. Most flexible but does require a bit more configuration and comfort with SQL

Problem 6.3: Server Sync (6 points)

Problem 6.3.1: Triggering Sync (2 points)

Let's say you wanted to evolve your application by allowing users to save their shopping lists to a server that you maintained for this application. You would want to trigger saving to the server periodically and display a green checkmark (all data saved remotely!) when the latest changes were synced. Could you safely add the server call to the block of code inside **addButton.setOnClickListener(v -> {** ? Why or why not?

RUBRIC
2 points - No! The Main Android Thread (UI Thread) handles the **onClick** callback and it should not make API/network calls because that may block it and render the rest of the application unresponsive.

Problem 6.3.2: Triggering Sync Improvements (2 points)

If your answer above was that making the server call in **addButton.setOnClickListener(v -> {** is not the best strategy, describe a better way for making that call

RUBRIC
Up to two points based on below:

- **1 point -** The call to the server needs to be performed on a different thread
- **1 point -** This can be accomplished by creating a new Thread in-line
- **1 point -** This can be accomplished by creating a new Handler and Looper
- **1 point -** This can be accomplished by creating a new Executor Service with its own thread pool

Problem 6.3.3: Updating the User (2 points)

Building on your responses above, and with the requirement of rendering a green checkmark and a "all data saved remotely!" message, how, when, and by whom would those UI elements be updated in response to the server receiving the latest state of the user's items?

RUBRIC
Up to two points based on below:

- **1 point -** When handing the work off to a different thread, callbacks for success and failure should be registered
- **1 point** - The success callback can hand work back to the Main (UI) thread via calling **runOnUIThread** method
- **1 point** - running code via **Handler(Looper.getMainLooper())**

Problem 7: Polymorphism, Composition & Inheritance (10 points)

You are building a fairly straightforward, two dimensional, Android game. The game involves the main player, some enemies, and some other objects that the player and enemies can interact with. You have defined the following interface that every entity/object that is part of the game is going to implement.

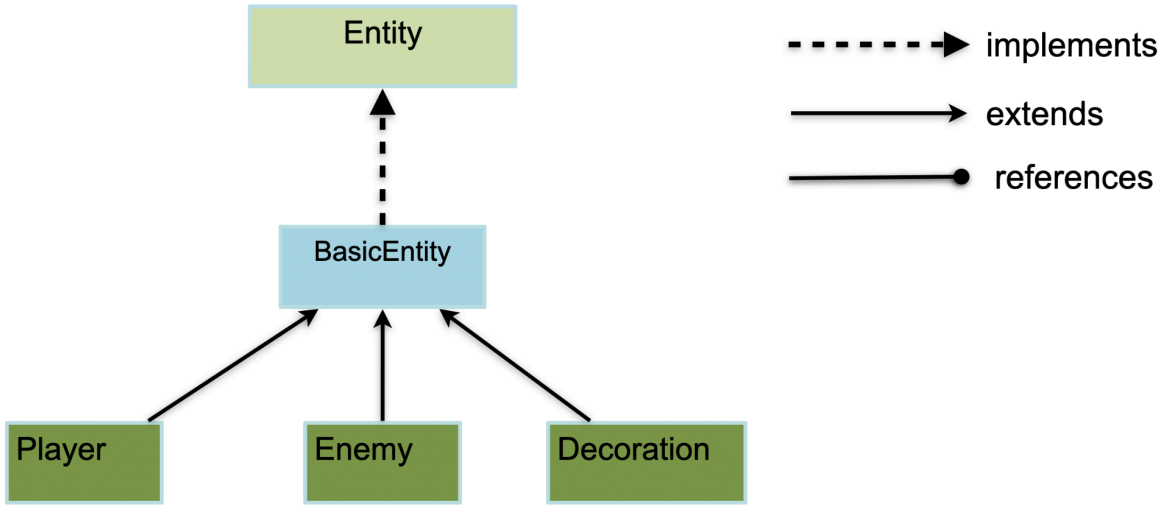
```
/**
 * Defines basics for all objects in our game.
 */
public interface Entity {
    /**
     * Moves the game object along the x and y axes.
     */
    void move(int x, int y);

    /**
     * Renders the entity onto the screen.
     */
    void render(GameScreen gameScreen);

    /**
     * Collide with another entity and optionally return a new resulting
combined entity.
     */
    Entity collide(Entity otherEntity);
}
```

Problem 7.1: Which one is it? (4 points)

You have started with the following class diagram:



The **BasicEntity** class provides default implementations for **move, render,** and **collide**. Its child classes can override those default behaviors as needed.

Problem 7.1.1: Polymorphism (1 point)

Does this structure provide the ability to take advantage of polymorphism?

RUBRIC

- Yes! Code interacting with game entities can refer to variables as **Entity** without needing to know what the exact implementation is.

Problem 7.1.2: is-a or has-a ? (1 point)

Are the relationships between Player-BasicEntity, Enemy-BasicEntity, Decoration-BasicEntity examples of **is-a** or **has-a** relationships? Please explain in a sentence why

RUBRIC

- They are **is-a** relationships. Each of the child classes is-a **BasicEntity** in that it inherits all of their state and behaviors. So each child is a **BasicEntity** but potentially with more functionality or different functionality for something that the **BasicEntity** already does.

Problem 7.1.3: Composition or Inheritance ? (2 points)

Is the diagram above an example of composition or inheritance?

RUBRIC

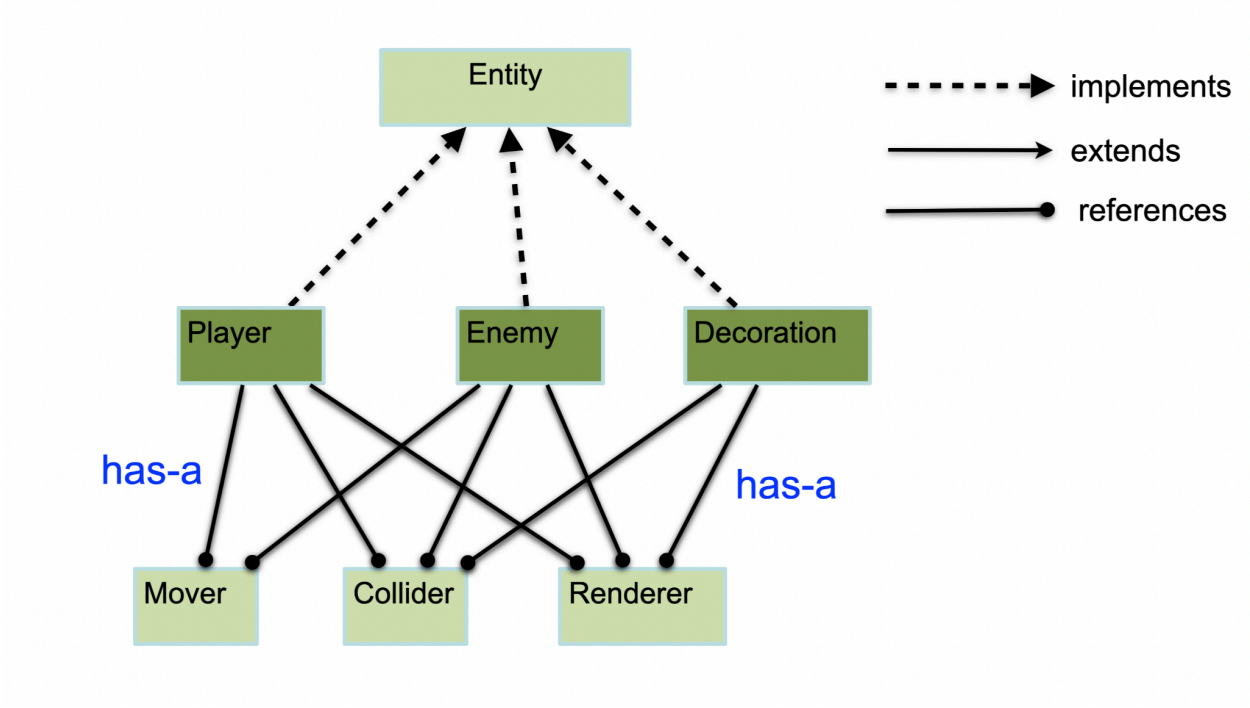
- This is an example of inheritance. BasicEntity provides some default implementations and the three classes below it **extend** it, thereby **inheriting** those implementations and have the option to override them.

Problem 7.2: An Alternative (2 points)

If your answer above was “This is Composition”, redraw the diagram as it would look like if it were “Inheritance”. If your answer above was “This is Inheritance”, redraw the diagram as it would look like if it were “Composition”.

RUBRIC

The example above was inheritance and this is what the class structure would look like with composition. Instead of inheriting common behavior, the concrete classes **Player, Enemy,** and **Decoration** would be composed with shareable implementations of the main pieces of functionality **move, collide,** and **render.**



Problem 7.3: Trade Offs (3 points)

Explain what the pros and cons of composition and inheritance are and when you would favor each.

RUBRIC
Up to three points based on

- **1 point** - Composition leads to better modularity because the lines between components are very clear.
- **1 point** - Composition leads to better encapsulation as there is no ambiguity about where in the parent-child hierarchy state is coming from and owned.
- **1 point -** Inheritance can lead to slightly less boilerplate code. This benefit is generally minor compared to the other two
- **1 point -** In general, composition is preferred and should be defaulted to. Inheritance can be chosen for straightforward use cases where the parent-child hierarchy is bounded and modularity/encapsulation are not significantly sacrificed.

Problem 7.4: Dependency Injection (1 point)

Explain in which scenario above might dependency injection come into play. How would it be used and what would be its benefit?

RUBRIC
At most one point for any of the following:

- **1 point -** Dependency injection pushes the benefits of polymorphism and composition to the extreme by enabling collaborators to have no references to concrete implementations of the classes that they are composed with
- **1 point -** Collaborators are passed in at construction time by a third party that orchestrates the composition between collaborators.