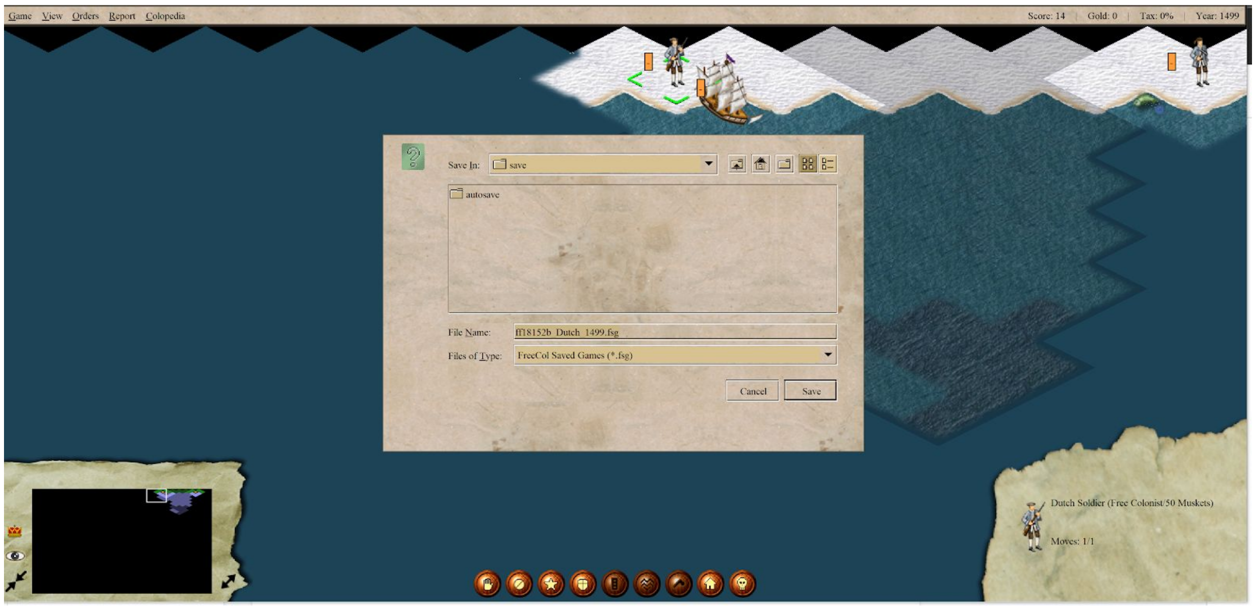
Save Game Feature

Saving the game is an essential feature because if the player has to stop his current gaming session, the state of the game can be stored. The player will then be able to load the current state of the game in the feature when he wishes to continue playing the game. Below are all the classes and methods necessary to make this feature operate according to the FreeCol game guide.



Save In: save

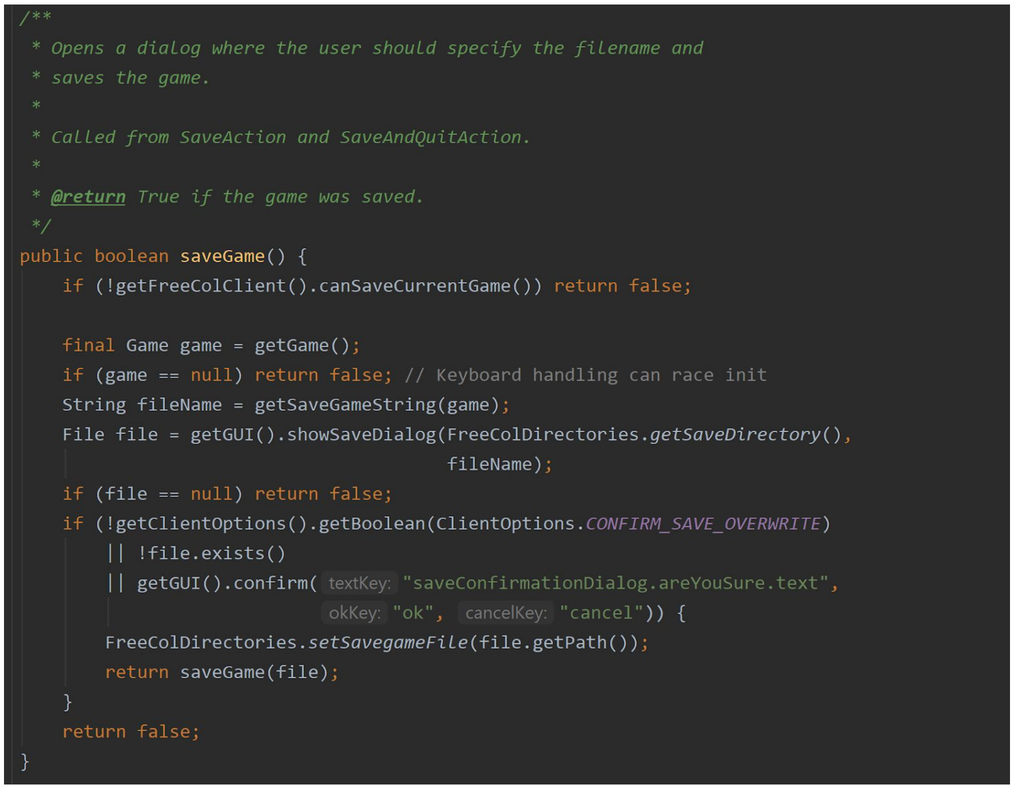
autosave

File Name: ff18152b Dutch 1499.fsg

Files of Type: FreeCol Saved Games (*.fsg)

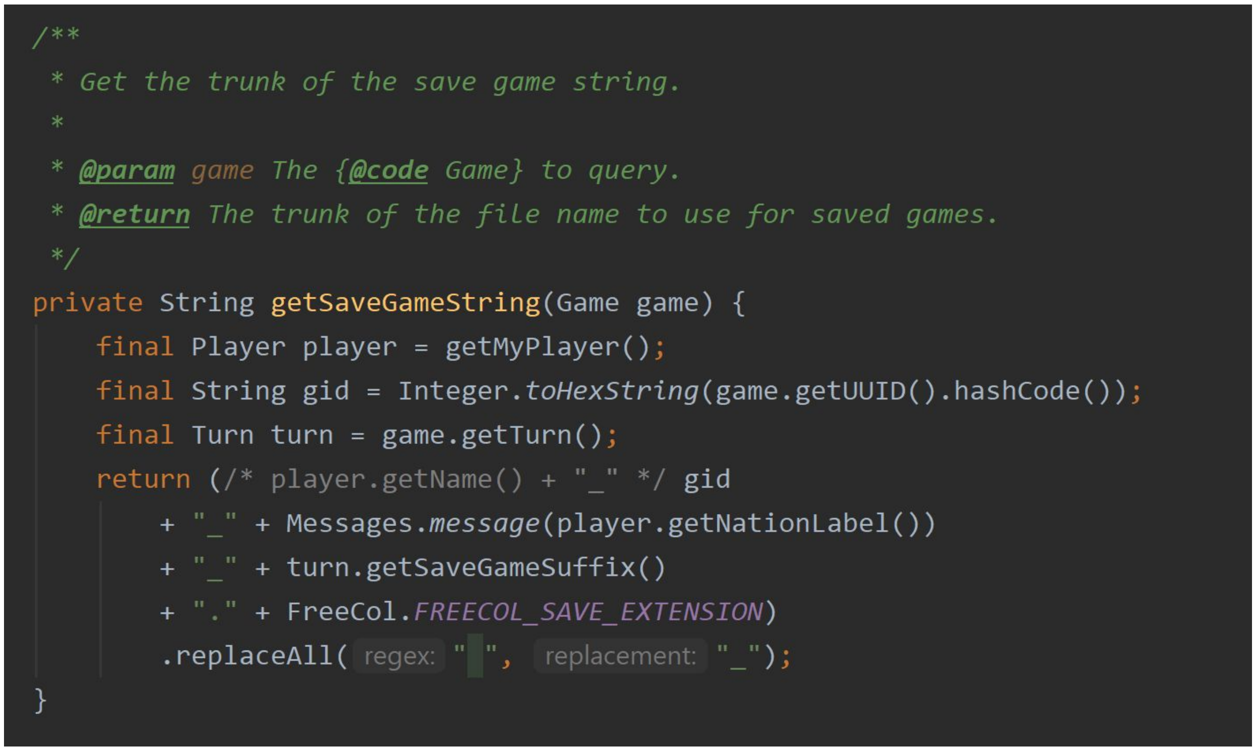
Client Side

The “saveGame” method in the InGameController class is called whenever the user requests the game to be saved. This method receives no parameters and outputs a boolean that indicates if the game was successfully saved or not. First, it checks if the game can be saved and if there is an active and valid game session.



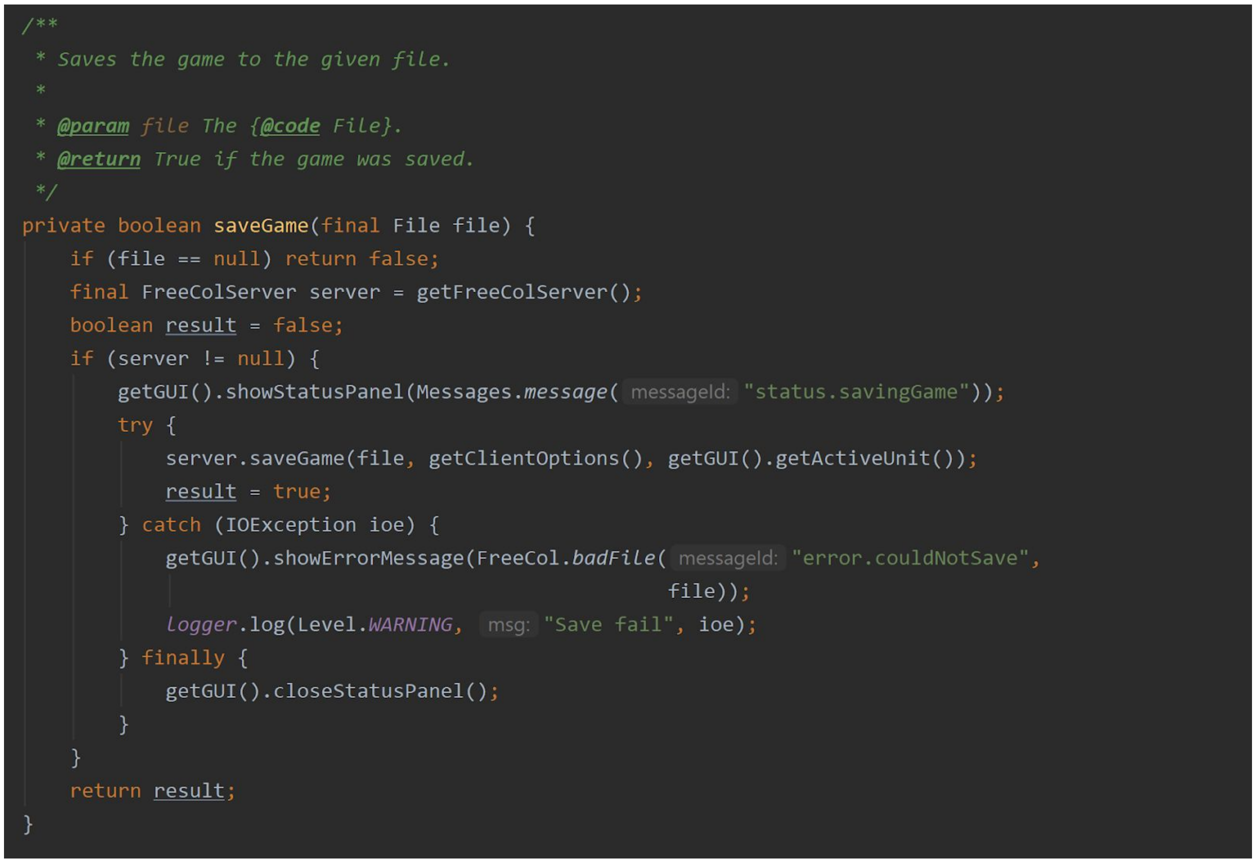
```java
/**
 * Opens a dialog where the user should specify the filename and
 * saves the game.
 *
 * Called from SaveAction and SaveAndQuitAction.
 *
 * @return True if the game was saved.
 */
public boolean saveGame() {
    if (!getFreeColClient().canSaveCurrentGame()) return false;

    final Game game = getGame();
    if (game == null) return false; // Keyboard handling can race init
    String fileName = getSaveGameString(game);
    File file = getGUI().showSaveDialog(FreeColDirectories.getSaveDirectory(),
                                        fileName);
    if (file == null) return false;
    if (!getClientOptions().getBoolean(ClientOptions.CONFIRM_SAVE_OVERWRITE)
        || !file.exists()
        || getGUI().confirm( textKey: "saveConfirmationDialog.areYouSure.text",
                             okKey: "ok",  cancelKey: "cancel")) {
        FreeColDirectories.setSavegameFile(file.getPath());
        return saveGame(file);
    }
    return false;
}
```

“getSaveGameString” generates a name for the file that will be created. It uses the game’s unique id and the player’s in-game preferences to generate it.



```java
/**
 * Get the trunk of the save game string.
 *
 * @param game The {@code Game} to query.
 * @return The trunk of the file name to use for saved games.
 */
private String getSaveGameString(Game game) {
    final Player player = getMyPlayer();
    final String gid = Integer.toHexString(game.getUUID().hashCode());
    final Turn turn = game.getTurn();
    return (/* player.getName() + "_" */ gid
        + "_" + Messages.message(player.getNationLabel())
        + "_" + turn.getSaveGameSuffix()
        + "." + FreeCol.FREECOL_SAVE_EXTENSION)
        .replaceAll( regex: " ",  replacement: "_");
}
```

A file browser dialog is then open so the user can choose where to save the game. The user might be asked if the file is to be overwritten if it already exists, if so he must confirm that he wants to overwrite. In the end, the file is created and the file path is stored locally for further use through the use of the “setSaveGameFile” method. Then the other “saveGame” method is invoked.

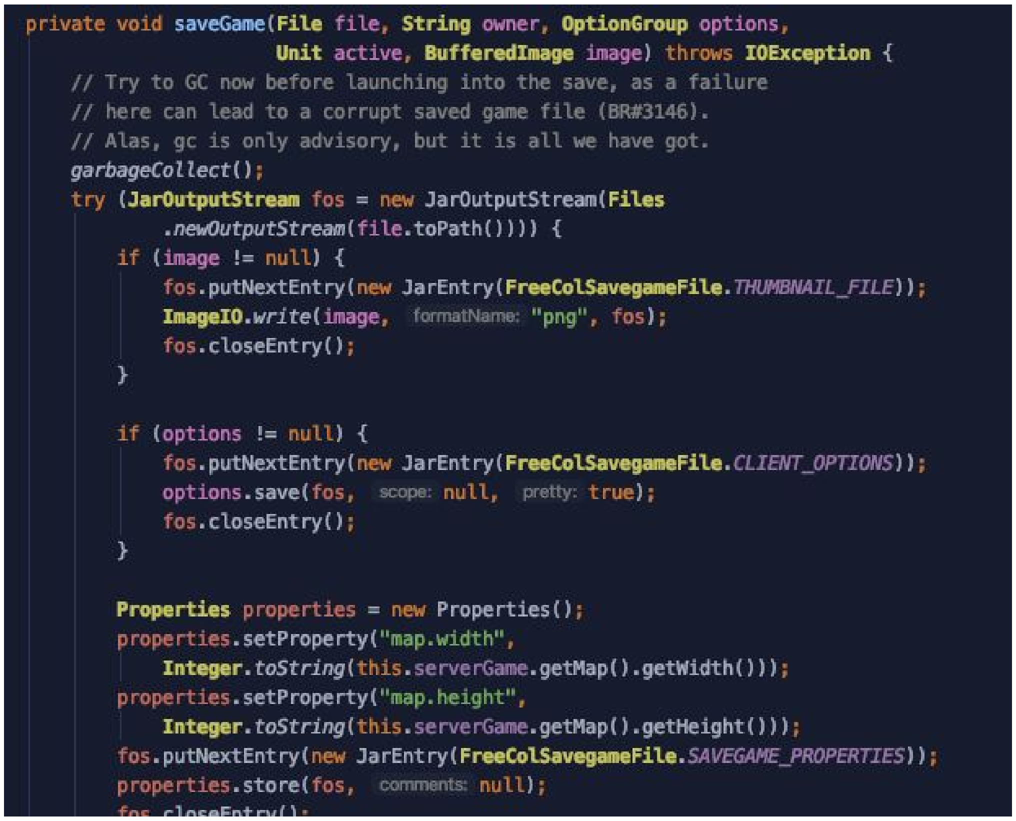


```java
/**
 * Saves the game to the given file.
 *
 * @param file The {@code File}.
 * @return True if the game was saved.
 */
private boolean saveGame(final File file) {
    if (file == null) return false;
    final FreeColServer server = getFreeColServer();
    boolean result = false;
    if (server != null) {
        getGUI().showStatusPanel(Messages.message( messageId: "status.savingGame"));
        try {
            server.saveGame(file, getClientOptions(), getGUI().getActiveUnit());
            result = true;
        } catch (IOException ioe) {
            getGUI().showErrorMessage(FreeCol.badFile( messageId: "error.couldNotSave",
                                                       file));
            logger.log(Level.WARNING,  msg: "Save fail", ioe);
        } finally {
            getGUI().closeStatusPanel();
        }
    }
    return result;
}
```

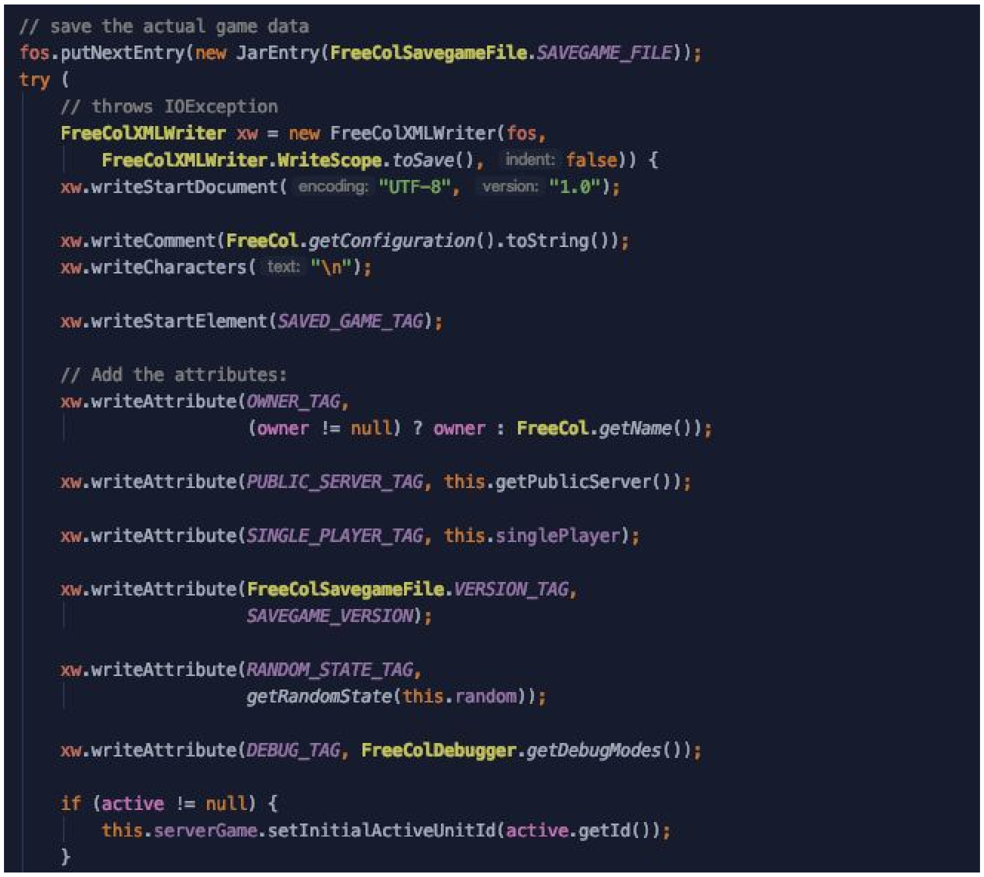
This method receives the save file as input and outputs a boolean that indicates if the game was able to be saved or not. This method checks if the server object is declared, throwing an exception if it is not. The client requests the server to save the game by passing the save file, the client options specific to the player (these are the current player’s preferences), and the current active unit.

Server Side

Once the game has been saved on the client side, the saveGame method on the FreeColServer file is called. A JarOutPutStream is created to save the properties of the game - thumbnail, client_options, and server game/map. Then the contents of the JarOutPutStream are added into an XML file through the FreeColXMLWriter. It is through this FXML writer that the game is able to load the data for future play.



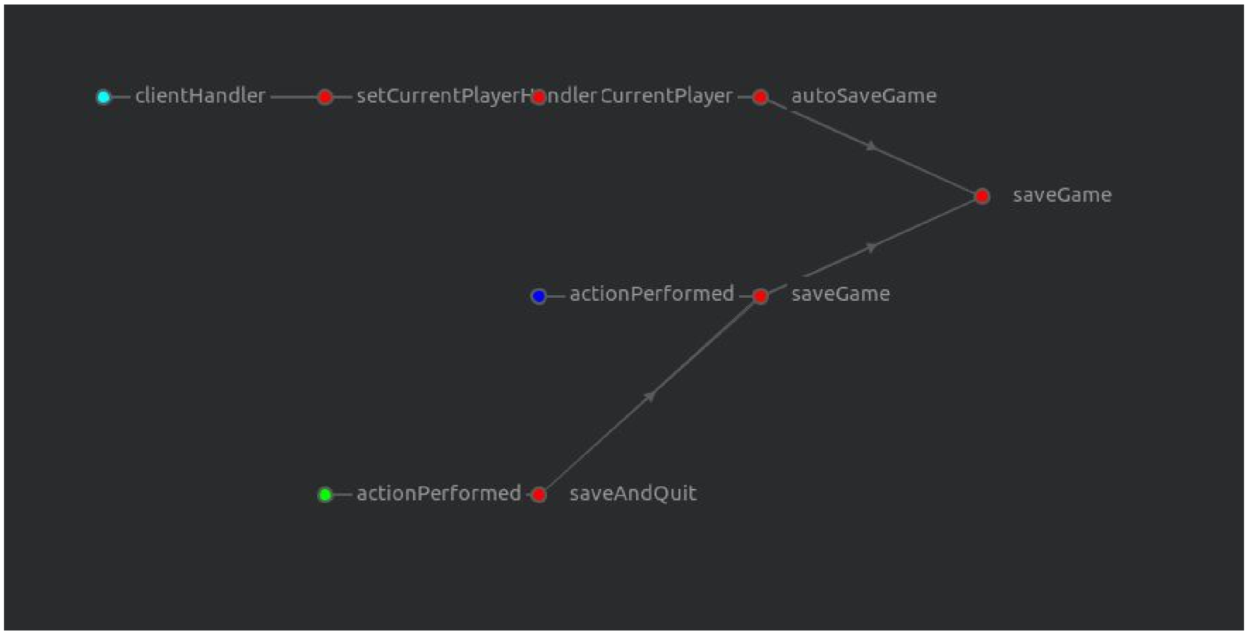
```java
private void saveGame(File file, String owner, OptionGroup options,
                      Unit active, BufferedImage image) throws IOException {
    // Try to GC now before launching into the save, as a failure
    // here can lead to a corrupt saved game file (BR#3146).
    // Alas, gc is only advisory, but it is all we have got.
    garbageCollect();
    try (JarOutputStream fos = new JarOutputStream(Files
            .newOutputStream(file.toPath()))) {
        if (image != null) {
            fos.putNextEntry(new JarEntry(FreeColSavegameFile.THUMBNAIL_FILE));
            ImageIO.write(image,  formatName: "png", fos);
            fos.closeEntry();
        }

        if (options != null) {
            fos.putNextEntry(new JarEntry(FreeColSavegameFile.CLIENT_OPTIONS));
            options.save(fos,  scope: null,  pretty: true);
            fos.closeEntry();
        }

        Properties properties = new Properties();
        properties.setProperty("map.width",
            Integer.toString(this.serverGame.getMap().getWidth()));
        properties.setProperty("map.height",
            Integer.toString(this.serverGame.getMap().getHeight()));
        fos.putNextEntry(new JarEntry(FreeColSavegameFile.SAVEGAME_PROPERTIES));
        properties.store(fos,  comments: null);
        fos.closeEntry();
```



```java
// save the actual game data
fos.putNextEntry(new JarEntry(FreeColSavegameFile.SAVEGAME_FILE));
try (
    // throws IOException
    FreeColXMLWriter xw = new FreeColXMLWriter(fos,
        FreeColXMLWriter.WriteScope.toSave(),  indent: false)) {
    xw.writeStartDocument( encoding: "UTF-8",  version: "1.0");

    xw.writeComment(FreeCol.getConfiguration().toString());
    xw.writeCharacters( text: "\n");

    xw.writeStartElement(SAVED_GAME_TAG);

    // Add the attributes:
    xw.writeAttribute(OWNER_TAG,
                      (owner != null) ? owner : FreeCol.getName());

    xw.writeAttribute(PUBLIC_SERVER_TAG, this.getPublicServer());

    xw.writeAttribute(SINGLE_PLAYER_TAG, this.singlePlayer);

    xw.writeAttribute(FreeColSavegameFile.VERSION_TAG,
                      SAVEGAME_VERSION);

    xw.writeAttribute(RANDOM_STATE_TAG,
                      getRandomState(this.random));

    xw.writeAttribute(DEBUG_TAG, FreeColDebugger.getDebugModes());

    if (active != null) {
        this.serverGame.setInitialActiveUnitId(active.getId());
    }
```

Usage (Upstream Diagram)

The saveGame function is called from three different places: autoSaveGame() and saveAndQuit() in client/control/InGameController, and the SaveAction class, which is used in the in-game menu.


clientHandler
setCurrentPlayerHandler
CurrentPlayer
autoSaveGame
saveGame
actionPerformed
saveGame
actionPerformed
saveAndQuit