DEVCOM DAC-TR-2021-007
January 2021

The Soldier and Squad Trade Space Analysis Framework (SSTAF)

by Ronald A. Bowers

DISTRIBUTION STATEMENT A. Approved for public release; distribution is unlimited.

DESTRUCTION NOTICE

Destroy by any method that will prevent disclosure of contents or reconstruction of the document.

DISCLAIMER

The findings in this report are not to be construed as an official Department of the Army position unless so specified by other official documentation.

WARNING

Information and data contained in this document are based on the input available at the time of preparation.

TRADE NAMES

The use of trade names in this report does not constitute an official endorsement or approval of the use of such commercial hardware or software. The report may not be cited for purposes of advertisement.

DEVCOM DAC-TR-2021-007
January 2021

The Soldier and Squad Trade Space Analysis Framework (SSTAF)

by Ronald A. Bowers
DEVCOM Data & Analysis Center

DISTRIBUTION STATEMENT A. Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED *(From - To)*
January 2021	Technical Report	February 2019–September 2020

4. TITLE AND SUBTITLE
The Soldier and Squad Trade Space Analysis Framework (SSTAF)

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Ronald A. Bowers

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Director
U.S. Army DEVCOM Data & Analysis Center
6896 Mauchly Street
Aberdeen Proving Ground, MD 21005-5071

8. PERFORMING ORGANIZATION REPORT NUMBER

DEVCOM DAC-TR-2021-007

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The U.S. Army is seeking to accelerate the development of systems to counter near-peer adversaries in a multi-domain environment. To do so, the Army plans to leverage modeling and simulation to help guide its research, development, engineering and acquisition efforts. One particularly difficult modeling problem is estimating Soldier performance in a combat environment. To provide the required capability, the U.S. Army Combat Capabilities Development Command Data & Analysis Center is developing the Soldier and Squad Trade Space Analysis Framework (SSTAF). SSTAF is a software infrastructure system for integrating multiple human performance and other models to provide a unified representation of Soldier state, capability and behavior. SSTAF models the Soldier as a system, where the results of one model can affect the results of other models, and both the positive and negative effects of Soldier equipment can be captured.

15. SUBJECT TERMS
Soldier performance, squad performance, Soldier lethality, modeling and simulation

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Ronald A. Bowers
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAME AS REPORT	119	19b. TELEPHONE NUMBER *(include area code)* (410) 278-3348

Table of Contents

Table of Contents

List of Figures

List of Code Listings

Acknowledgements

The U.S. Army Combat Capabilities Development Command Data & Analysis Center recognizes the following individuals for their contributions to this report:

The author is

Ronald Bowers, DEVCOM Data & Analysis Center

The author wishes to acknowledge the contributions of the following individuals for their assistance in the creation of this report:

Timothy Myers, DAC

Gregory Dietrich, DAC

Dr. Karim Abdel-Malek, University of Iowa

Dr. Rajan Bhatt, University of Iowa

Executive Summary

This report documents the Soldier and Squad Trade Space Analysis Framework (SSTAF). SSTAF is a software infrastructure system for integrating multiple human performance and other models to provide a unified representation of Soldier state, capability and behavior. SSTAF models the Soldier as a system where the behavior of one model can affect the behavior of other models, and both the positive and negative effects of Soldier equipment are represented. The ultimate goal of SSTAF is to provide an architecture that enables the development of digital twins for specific Soldiers.

The report details the architecture of the SSTAF software and the technology used to develop the system, and provides multiple examples to explain how to integrate human performance and other models into SSTAF and how to integrate SSTAF into higher-level systems such as force-level models. It also discusses the models that have been developed or adapted to work with SSTAF as well as planned future efforts.

1. INTRODUCTION

In this report, I document the architecture of the Soldier and Squad Trade Space Analysis Framework (SSTAF). SSTAF is a software infrastructure system for integrating multiple human performance and other models to provide a unified representation of Soldier state, capability and behavior. SSTAF models the Soldier as a system, where the results of one model can affect the results of other models, and both the positive and negative effects of Soldier equipment can be captured. The ultimate goal of SSTAF is to provide an architecture that enables the development of digital twins for specific Soldiers. These digital twins can be used not only for material trade space analysis but also for interactive training and mission planning.

The key capabilities of SSTAF are the following:

- Model Soldier state and capability, update the state according to simulation events and modify the behavior of integrated models according to the current state.
- Support flexible anthropometric, human performance and equipment configurations to enable modeling at multiple levels of resolution to include modeling specific individual Soldiers.
- Provide an extensible application programming interface (API) usable for both interactive systems and force-on-force models.

Figure 1 shows an overview of the SSTAF concept. The box at the bottom shows the SSTAF system. Multiple models that predict various aspects of the Soldier can be loaded into the framework. SSTAF models can depend on other models. SSTAF provides mechanisms to reconcile dependencies between models and enable models to update each other according to what has occurred in the simulation.

The upper box shows how SSTAF can be integrated into client applications to fulfill user-specific purposes. This aspect of SSTAF illustrates one of its primary merits. SSTAF provides model developers with a single target for integration. Rather than having to integrate models into multiple environments, models can be integrated into SSTAF and SSTAF provides the integration into higher-level constructs. This approach provides two significant benefits. First, it reduces development costs and repeated work since both models and client applications can program to stable SSTAF interfaces. Second, this approach helps ensure that human performance and behavior are represented consistently across different environments.

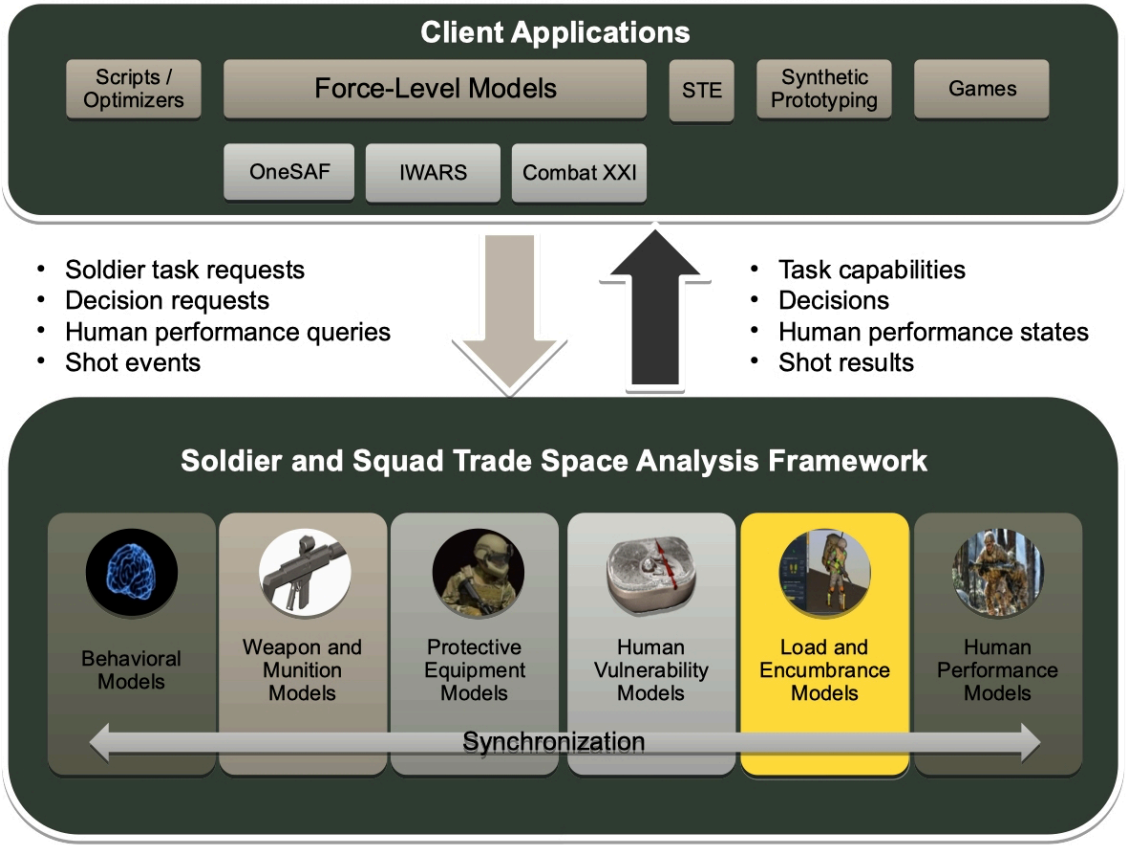

Figure 1. SSTAF concept

1.1 Motivation

SSTAF was inspired by the stated requirements of the Soldier Lethality (SL) Cross Functional Team (CFT). These requirements were specified in 2018 through the SL CFT modeling and simulation strategy and were reaffirmed in August 2020. The strategy is expressed by the following vision and objective:

- **Vision Statement**
  Develop a modeling and simulation system that is able to provide timely, affordable trade space analysis at the Squad level with enough fidelity to enable decision-making.
- **Long-Term Objective**
  Develop a squad modeling and simulation system with a single front-end user interface, preloaded with multiple approved vignettes and capable of running closed form, to conduct trade analysis. Update the system with emerging technologies/techniques (friendly and adversary) as they emerge.

I designed SSTAF to enable the vision by providing the mechanisms to model the Soldier with enough fidelity to enable decision-making. The CFT's long-term objective to develop a complete trade space analysis environment will be realized by the *Squad Operational Value*

Evaluation using Realistic Metrics and Tactical Capability Hierarchies (*Squad OVERMATCH*) system. Squad OVERMATCH is discussed in Section 7.3.

1.2 Purpose and Organization of the Report

This report serves two primary purposes. First, it documents the architecture of SSTAF Version 1.0 to help those who might maintain and extend it. Second, it instructs developers on how to develop SSTAF-compliant models and integrate SSTAF into higher-level models and applications.

The report comprises four parts. The first part consists of Sections 2 and 3, which discuss the architecture of the SSTAF system and the technologies that I used to enable it. In the second part, Sections 4 and 5, I use multiple examples to explain how to implement SSTAF-compliant models and build SSTAF-based applications. In the third part, Section 6, I briefly discuss some of the models that have been adapted and integrated into SSTAF. In the final part, Section 7, I discuss planned future directions for the project.

2. TECHNOLOGIES

In this section, I describe the technologies that I used to implement SSTAF. In general, I chose these technologies to enable quick and low-risk development of the core system architecture.

2.1 Java

I implemented the SSTAF core system in Java. I chose Java for several reasons. First, I am most fluent in Java and can implement concepts very rapidly using the Java platform. Second, integrated development environments (IDEs) for Java enable rapid development and easy refactoring. This is particularly important early in development when concepts, architectures and APIs are most likely to change, as they did several times in the early stages of development. Next, the Java continuous integration tool chain is mature and well supported in cloud environments like DI2E and the U.S. Army Futures Command Modernization Application and Data Environment (MADE). Two of the three combat simulations targets for integration, specifically One Semi-Automated Force (OneSAF) and Combat XXI, are implemented in Java, so integration with those systems will be easier. Finally, after a period of stagnation, the Java platform is now receiving frequent improvements in response to developer needs. These improvements include ahead-of-time compilation to native applications and libraries.

I implemented and tested SSTAF Version 1.0 using Java 11. I chose this version because it is the first long-term support version that includes the features upon which SSTAF relies. I discuss those features in the following two sections. An additional reason for using Java 11 is that it is the version that OneSAF is currently using. Since OneSAF will be the first integration target for SSTAF, it is appropriate to run on the same platform.

2.1.1 Java Module System

SSTAF makes extensive use of the Java Module System (JMS). JMS is an enhancement to the Java language and platform that facilitates the development of reusable software components.

Introduced in Java 9 as a way to shrink the size of Java applications, JMS establishes the module as a first-class Java language and platform construct. Modules comprise one or more packages that are bundled together in a single jar file along with a **`module-info.java`** file that specifies the properties of the module.

Through the **`module-info.java`** file, the module can specify which packages the module exports, that is, made visible to other modules. This provides a layer of visibility control beyond the private, package and public visibility specifications of earlier Java versions. The effect is that classes within the module that are declared public but are not in an exported package are effectively private to the entire module. Such classes can be accessed freely from within the

module but are not accessible from other modules. This extra level of access control facilitates building modules with narrow, well-defined interfaces that hide their internal implementations.

Modules can declare dependencies on other modules. This is done using a **`Requires`** specification in the **`module-info.java`** file. If a module specifies a requirement for another module and that module is not found when the referring module is loaded, the virtual machine will throw an error and exit. Modules are made accessible to an application by placing them on the module path. The module path is the module-oriented successor to the legacy Java class path.

The JMS aligns strongly to the Java service paradigm and SSTAF utilizes this alignment to enable dynamic loading of models and other features. Modules can declare that they implement a service interface by providing a **`provides`** statement in the **`module-info.java.`** This support for service architectures enables an application to use a **`ServiceLoader`** to find implementations of desired interfaces among the modules provided on the module path. SSTAF leverages the service paradigm to enable the dynamic loading of models and flexible assembly of Soldier configurations. The details of this mechanism are discussed in Section 3.4.

2.1.2 Method References and Lambdas

SSTAF use two additional recently added Java features. These are method references and lambdas. Both features were introduced in Java 8. The primary motivation for the addition of these features was to support stream processing; however, both are very helpful in other use cases.

Method references are approximately equivalent to function pointers in C and C++. They specify the name and the containing object of a method and they can be passed as parameters to other methods. The most common use of method references in SSTAF is in input parsers where they are used to reference setters in builder objects. This construct is described in more detail in Section 3.3.1.

Lambdas are anonymous blocks of code that can also be passed as parameters to methods. Lambdas have the additional feature of having read access to variables in the code block in which the lambda is defined. Thus, lambdas can be passed as snapshots of the object state to other methods. The most common use of lambda expressions in SSTAF is as the argument to internal iteration methods such as **`Collections.forEach()`**.

2.2 JavaScript Object Notation

SSTAF uses JavaScript Object Notation (JSON) for its configuration files. I chose JSON for this role because it is relatively easy for a human to read and write, especially in an IDE.

Furthermore, it requires much less ceremony, that is, fewer superfluous symbols and tags than Extensible Markup Language (XML).

To simplify processing JSON configuration files, SSTAF includes the **`JSONUtilities`** class. **`JSONUtilities`** includes numerous static methods for loading JSON files and processing the contents. One particularly useful feature of **`JSONUtilities`** is its ability to handle file references in place of embedded objects. This enables the user to split analysis inputs into smaller files and reuse them rather than embedding numerous copies of the same information. For example, the configuration for an M4 carbine and its magazines can be defined once and included by file reference in each Soldier. Code Listing 1 shows an extracted section from a SSTAF demonstration input file and demonstrates mixing file references and in-line object definitions.

Code Listing 1. JSON Configuration File

```
{
  "configurations": {
    "Simple Anthropometry": "Anthropometry.json",
    "Kit Manager": "../common/StandardKit.json",
    "Dynamic Aim": "dynamicAimConfig.json",
    "Telemetry Agent": {
      "statesToRecord": [
        "Aim Internals",
        "Muscle Metrics"
      ]
    }
}
```

Note that although SSTAF-compliant models are not required to use JSON for configuration, as part of the model initialization sequence, SSTAF will attempt to provide a JSON configuration to the model. Therefore, if the model uses a different configuration mechanism, it is best practice that the model support receiving at least the filename for the model-specific configuration file through the JSON configuration mechanism. This will enable a single input file graph to configure the SSTAF system and models. The details of the configuration mechanism are discussed in Section 3.4.

2.3 Gradle

SSTAF uses Gradle for its build system. I chose Gradle was over Maven, because it is somewhat more flexible and faster. However, the main advantage is that Gradle uses Groovy for its configuration language whereas Maven uses XML. The Appendix provides instructions on how to obtain and build the SSTAF source code.

3. ARCHITECTURE OVERVIEW

This section describes the details of the SSTAF architecture and consists of four parts. The first part describes the requirements and goals that guided the development of the architecture. The second is a high-level overview of the architecture and its key concepts. The third part drills down to describe the details of the architecture, including model implementation and configuration. The fourth section describes other topics such as stochastic analysis and its corollary, repeatability.

Unified Modeling Language (UML) diagrams are used throughout this and later sections to represent elements of the architecture. UML provides a standard way to represent object-oriented systems like SSTAF as well as the processes that are executed on the system. A simple reference for UML can be found at http://holub.com/uml/.

3.1 Requirements for the SSTAF Architecture

The SSTAF concept shown in Section 1 reveals several requirements for the architecture. From the conceptually architecture, it is apparent that the system is required to do the following:

- represent a Soldier,
- enable multiple models of various aspects of Soldier state and capability to be integrated into the system,
- synchronize Soldier state and capability between models, and
- interact with different types of client applications.

In addition to these requirements, another requirement arose during early conversations with the Soldier performance community. This requirement was that models simply "plug into" SSTAF rather than be hard-wired into it. This requirement was intended to allow models to be developed independently from the framework and eliminate the need for U.S. Army Combat Capabilities Development Command Data & Analysis Center to manage integration of all of the models.

Solely in themselves, these requirements were not sufficient to force architectural decisions. Additional conversations with the community, analysis and experience with previous systems led to the derivation of more detailed requirements that did enable development. The derived requirements for SSTAF include the following:

- Dynamic (i.e., runtime) extensibility
- Clean separation of the core system from the models
- Simple and stable plugin and application interfaces
- The ability for models to interact with other models
- The ability to represent both Soldiers and noncombatants

- The ability to aggregate Soldiers into units and units into hierarchies to enable Squad- and higher-level models and performance metrics
- The ability to scale the analysis to large numbers of Soldiers to support force-on-force models
- The ability to exploit multi-core, multi-CPU and multi-computer (cloud and high-performance computing) systems to maximize performance
- Repeatable analyses
- Support for distributed development of both models and the core system
- Robust verification infrastructure

With the exception of multi-computer deployment, SSTAF 1.0 meets all of these requirements. How each requirement is achieved is discussed throughout the report.

3.2 Overview and Key Concepts

The architecture of SSTAF-based systems is illustrated in Figure 2. Each system can be envisioned as comprising three levels. At the top level are the SSTAF-based applications. These include force-level models such as OneSAF, interactive visual simulation such as the Synthetic Training Environment and other custom applications. At the bottom level are the SSTAF-compliant models. These models respond to requests from the application and calculate the state and capabilities of the Soldiers or other humans in the simulation. The SSTAF system modules sit in the middle and provide the infrastructure for loading and using the models. The SSTAF system connects the models together and enables the client application to make requests from them.

The diagram in Figure 2 reflects several of the major system requirements. First, the layered separation between application, framework and models addresses the requirement for clean separation between the framework and models. The framework has no dependencies on the models and the models have a single dependency on the framework that enables them to work in the system. The diagram also shows that dynamic extensibility is achieved using runtime plugins and that models are able to interact with each other using arbitrary interfaces defined by the models. Furthermore, the diagram illustrates that the interfaces between the application, framework and plugin layers are simple and owned by the SSTAF framework. Since the framework has no dependencies on the models or the applications, these interfaces will be very stable.

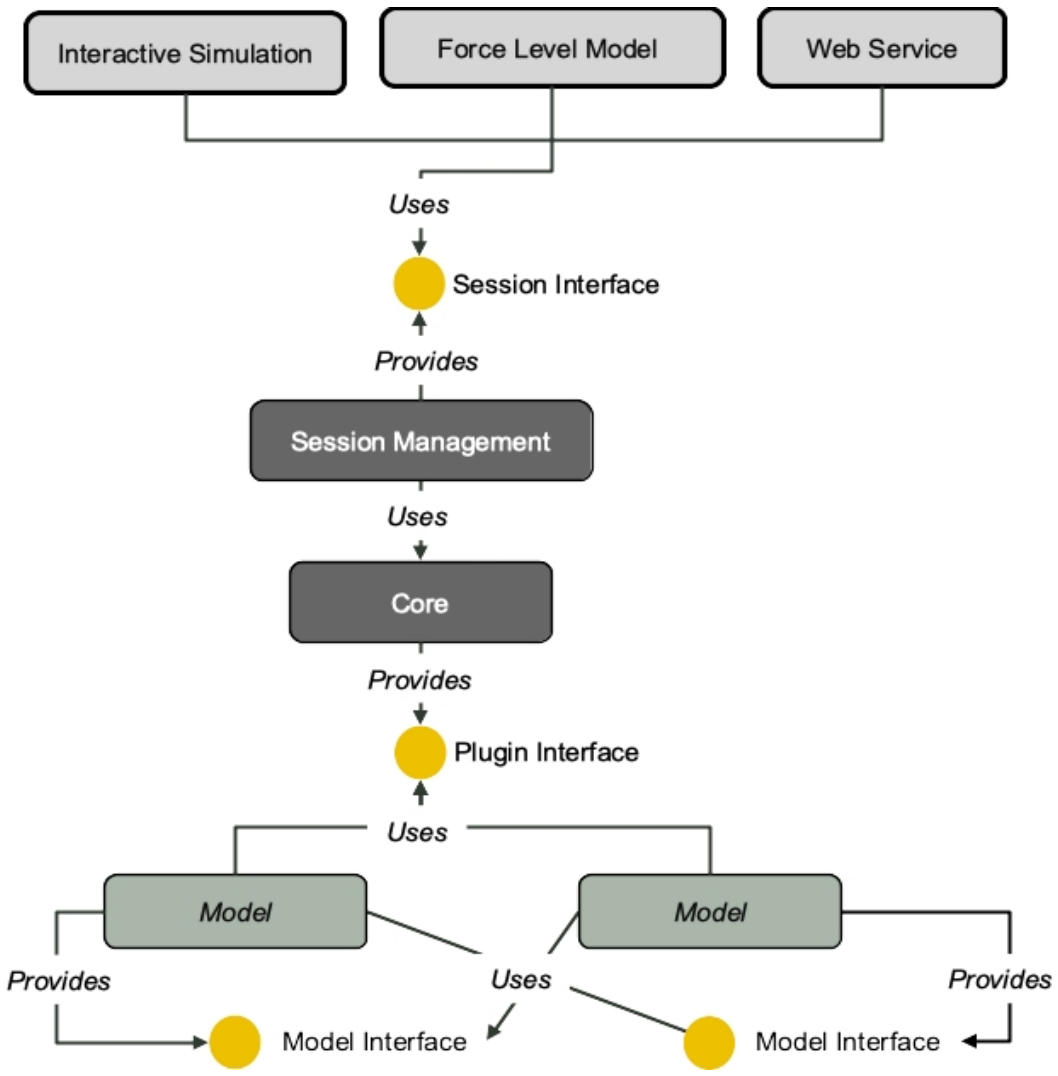

Figure 2. SSTAF system architecture

Some key aspects of the SSTAF system modules are that they are small and contain no sensitive information. This enables the SSTAF core to be shared with non-DOD organizations so that they can develop SSTAF-compliant models. Furthermore, because models are not hard-wired into SSTAF, SSTAF can be integrated into models like OneSAF without forcing the set of SSTAF-compliant performance models to be included with it. This makes distribution easier and facilitates distributed development of models as specified in the requirements. Finally, I implemented the SSTAF system using well-established technologies and an uncomplicated design. This reduced risk and enabled DEVCOM Data & Analysis Center to reach initial capability quickly. Thus, going forward, development effort can be concentrated on the models and the applications.

To organize the presentation of the architecture, the core concepts that guide the implementation of SSTAF and SSTAF-based systems are divided into two categories. The first category consists of the functional concepts. These concepts establish how objects in the analytic domain, such as weapons, humans, Soldiers and units, are represented. The second category consists of the structural concepts. This category describes how the SSTAF core, SSTAF-compliant models and SSTAF-based applications are constructed. It includes the architectural patterns that were used in the design. The structural and functional concepts overlap to define how SSTAF-compliant models such as those for mobility, injury or target detection are implemented.

3.2.1 Functional Concepts

There are three functional constructs that are central to SSTAF. These are entities, features and messages. Entities represent real-world things in SSTAF, specifically humans, Soldiers and units. Features implement models and add capabilities to entities. They are loaded into the system dynamically using plugins. Messages are used to communicate between individual entities or between the client environment and an entity. Features, specifically the **`Handler`** subtype, process messages when they are received by an entity. Figure 3 shows the **`Entity`** class hierarchy and its relationship to features and messages.

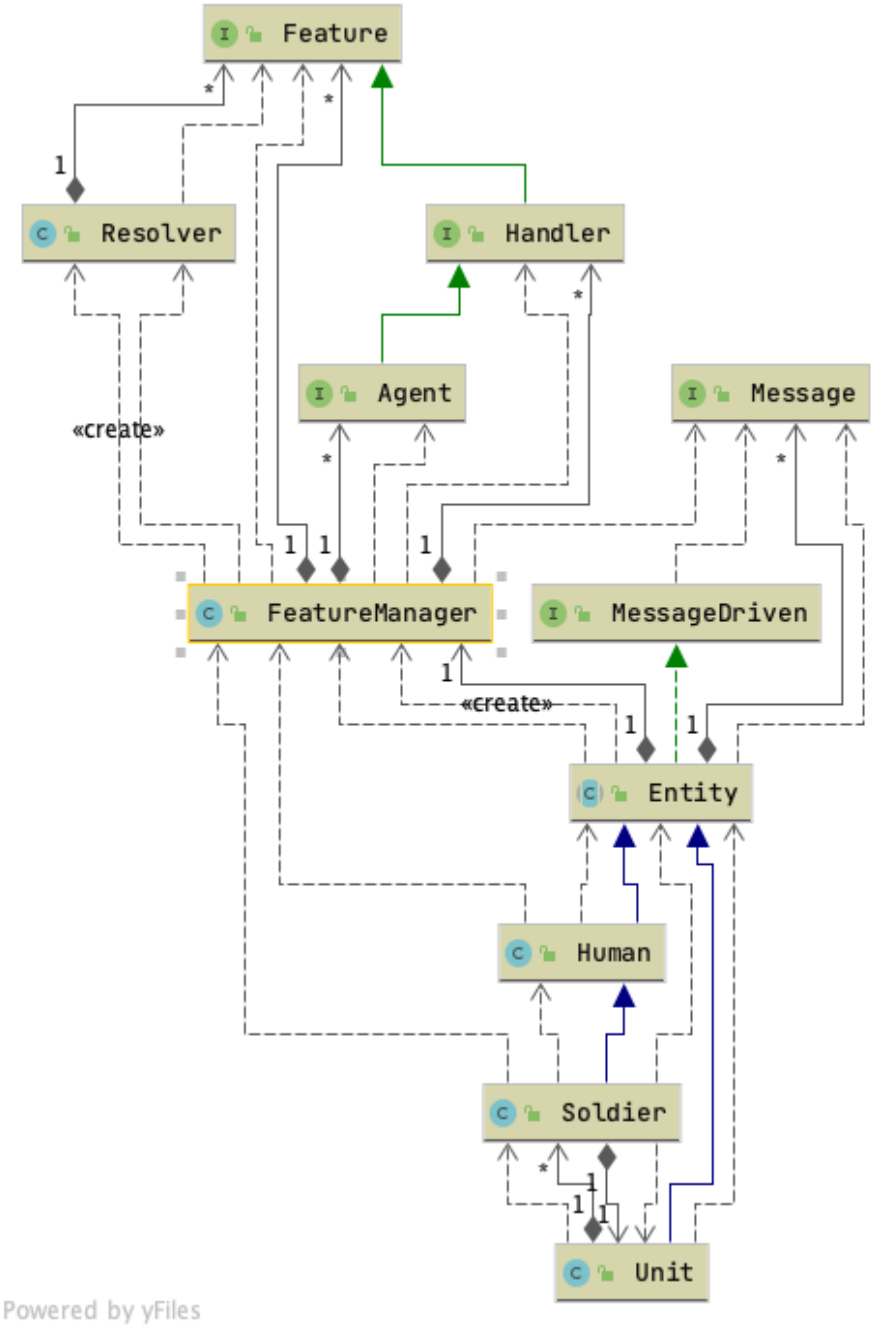

Figure 3. Entities and features

At the base of the **`Entity`** class hierarchy is the **`MessageDriven`** interface. **`MessageDriven`** defines the contract for objects that consume and emit messages. **`MessageDriven`** objects have two message queues, one for messages going into the object and the other for messages coming out.

Messages are objects passed between entities to issue a request for information or involve a command. The **`Message`** interface describes the essential behavior of a message object in SSTAF. A message object is described more precisely as an envelope because the important part of the message is not the outer class but its content. The content for a message can be of any type. This arrangement enables feature modules to add new types for communication without sub-classing **`Message`**.

Features are classes used to implement models or to add other capabilities to entities. They are implemented as plugins and packaged as JMS modules as described in Section 3.2. The features used by an entity during a run are specified in the analysis configuration files and added to the entity when the configuration files are read and the entity is created.

There are three types of features. The simplest feature type implements the base **`Feature`** interface. Implementing **`Feature`** enables a class to be loaded by the **`ServiceLoader`** and added to an **`Entity`**. The second type of feature implements the **`Handler`** interface. **`Handler`** extends **`Feature`** and adds the ability to receive, process and emit messages. The third type of feature is the **`Agent`**. The **`Agent`** interface extends **`Handler`** and adds a method that will be invoked on every tick of the simulation clock. The details of the **`Feature`**, **`Handler`** and **`Agent`** interfaces and their implementations are described in Sections 3.2.1, 3.2.2 and 3.2.3, respectively.

The **`Entity`** class is the base class for all simulation participants. It provides the core implementation for the **`MessageDriven`** architecture by providing the queues for receiving and returning messages and a mechanism for dispatching the messages to the appropriate loaded **`Handler`**. The **`Entity`** class and its subtypes are discussed in more detail in Section 3.3.

The programmatically declared interfaces for **`Entity`** and its descendants are simple. Entities are enhanced through composition rather than inheritance. They are enhanced at runtime by the features that have been added to the entity and the messages to which those features respond. The differences between **`Human`**, **`Soldier`** and **`Unit`** are small and are related solely to the ability to construct military hierarchies.

3.2.2 Structural Concepts

The core structural concepts employed in the SSTAF architecture might be apparent from the name of the project. SSTAF is a software *framework* that is enhanced by user-defined *plugins*.

A framework is a software scaffolding. It provides infrastructure and support services that facilitate further development but it is not itself a complete application. To produce a useful application or library, a framework must be extended. A key feature of a framework is that it defines the contract to which extensions must comply in order to work in the framework. Usually these contracts consist of an API that the extension must implement in order to be usable

in the framework. In the case of SSTAF, the required API is defined by the **`Feature`** interface hierarchy.

Frameworks can be extended either at compile time by adding code or linking in additional libraries, or at runtime by making dynamically loaded plugins available to the framework. For SSTAF, the plugin approach is used. Plugins are used to implement Soldier performance and other models as well as optional services such as telemetry services.

Plugins enable the models to be decoupled fully from the SSTAF core and for models and core to evolve at different rates. The plugin architecture allows SSTAF users to develop their own model suites and use them without changing the SSTAF core or impacting other SSTAF users. This arrangement also enables the SSTAF core to be configuration managed independently of the models or applications.

The SSTAF core is very small and divided into two modules. The first module, **`mil.sstaf.core`**, defines all of the core classes and interfaces. It also defines some common utilities for functions such as the **`JSONUtilities`** class for parsing and the dependency injection system. SSTAF plugin modules will depend on the core module and the model implementations that they provide must extend one of the **`Feature`** interfaces defined in the core module. The second module, **`mil.sstaf.session`**, defines the interface layer between client applications and the SSTAF environment. Package diagrams of the two modules are shown in Figures 4 and 5.

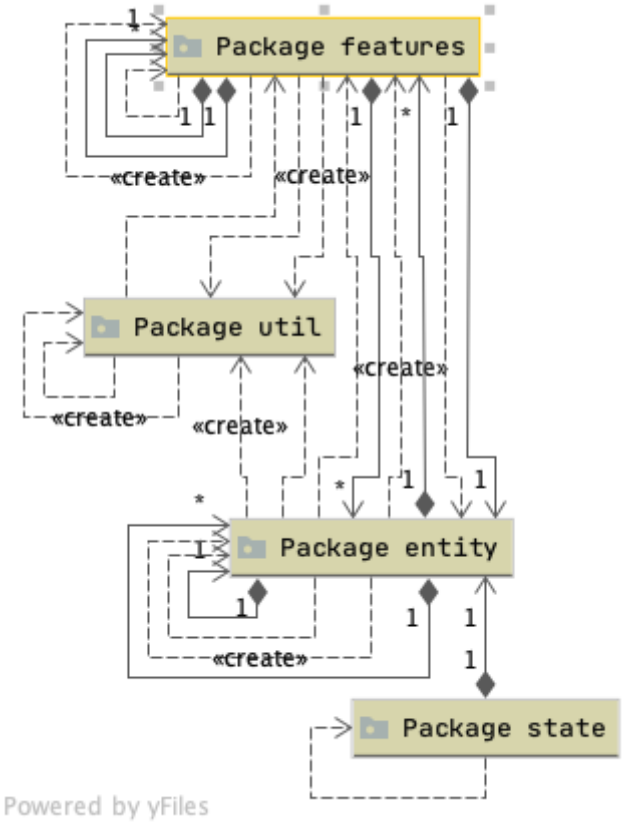

Figure 4. Core module

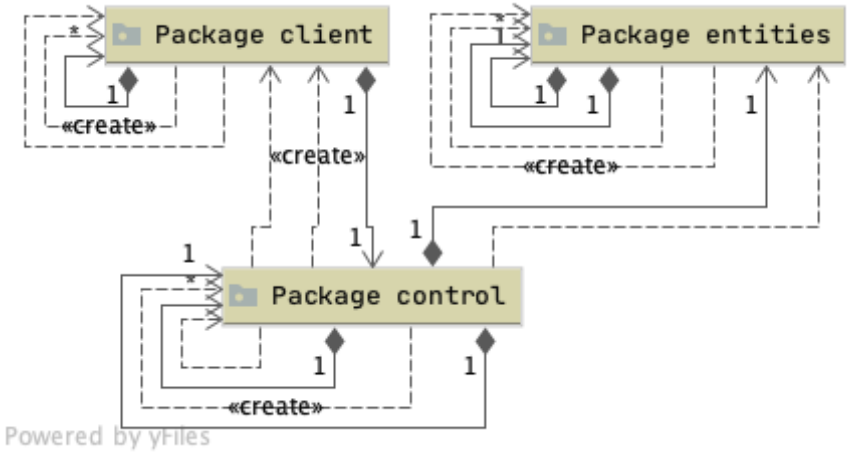

Figure 5. Session module

3.3 Features, Handlers and Agents

Features add capabilities to entities through dynamic composition. They implement the models that provide analytic capability to SSTAF. Features are loaded into the system at runtime based on configurations provided by the analyst or other subject-matter experts. Because the models are specified via configuration files rather than being hardwired into SSTAF, it is easy to change the models used in a simulation run. It is also possible to provide different models of the same phenomena. For example, different Soldiers could be configured to use different aim models.

A key capability of the feature system, indeed the basis for building interaction between models in SSTAF, is the ability of a feature to declare a dependency on other features. This dependency is expressed in the source code for the dependent **`Feature`** class by using a **`Requires`** annotation (see Section 3.3.5) and is resolved at runtime by a dependency-injection mechanism (see Section 3.5).

As shown in the class hierarchy in Figure 6, there are three types of features. These types are defined by the **`Feature`**, **`Handler`** and **`Agent`** interfaces. Each of the interfaces is partially implemented by an abstract base class, specifically **`BaseFeature`**, **`BaseHandler`** and **`BaseAgent`**.

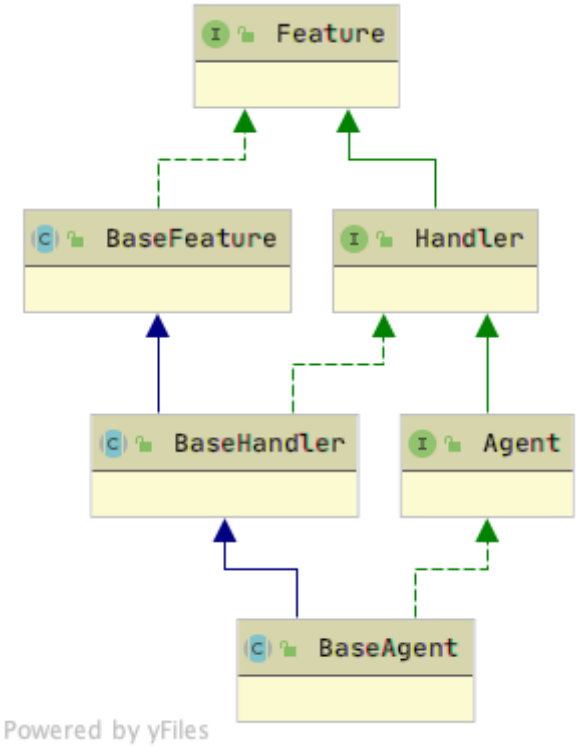

Figure 6. `Feature` class hierarchy

Although the primary purpose of the classes in the feature hierarchy is to add performance and other models to **`Humans`**, **`Soldiers`** and **`Units`**, implementations of **`Feature`** can be used to provide other functions. One example is the **`TelemetryAgent`** that can be added to any **`Entity`** to provide configurable logging of the state of the **`Entity`**. The **`TelemetryAgent`** is discussed in detail in Section 6.7.

Feature classes are specified and identified in SSTAF using a set of four parameters. These parameters are the name of the feature and its version number. The version number is expressed as major, minor and patch-level values. Although the feature name can be the class name, it is not required to be. Since these values are used in the SSTAF input files to specify which features to load, it can be helpful to give features names that can be easily understood by normal people, rather than obtuse names only understood by developers. The mechanisms for specifying, loading and adding features to entities are described in Section 3.5.

3.3.1 The `Feature` Interface

The **`Feature`** interface provides the API necessary for a plugin to be loaded into the system. Models and other plugins extend **`Feature`** to provide custom capabilities. The signature of methods of the **`Feature`** interface are shown in Figure 7.

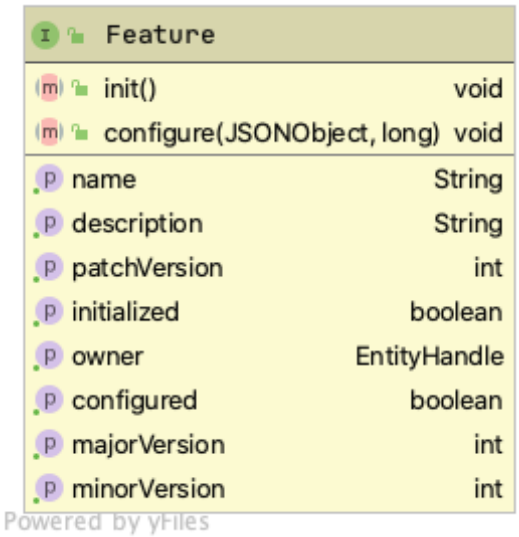

Figure 7. `Feature` interface

Four of the methods in the API are dedicated to providing the identification information for the **`Feature`**. These are the **`getName`**, **`getMajorVersion`**, **`getMinorVersion`** and **`getPatchVersion`** methods. These methods return the value for the feature name, major version, minor version and patch version of the plugin. The **`getDescription`** method provides a more verbose description of the feature. It is intended primarily to support interactive input preparation tools. The **`init`**, **`configure`**, **`isInitialized`** and **`isConfigured`** methods support preparing the **`Feature`** for use once it has been instantiated by the service loader and assigned to an **`Entity`**. The final method, **`getOwner`**, provides a reference to the **`Entity`** to which the **`Feature`** has been added.

The **`Feature`** interface can and should be extended by model developers to provide the desired functionality for their feature. Other features can interact with this feature using the extended interface. The mechanism for binding models together so that they can interact is discussed in Section 3.3.5.

Referring back to Section 2.2, note that the **`configure`** method takes two arguments. The first is a **`JSONObject`**. This object is the configuration information for the **`Feature`**. It is specified in the system input and provided to the **`Feature`** during startup. The second argument is the seed to be used if the **`Feature`** includes a random number generator.

Models and other plugins are free to extend the **`Feature`** interface in any way. However, developers will likely find it convenient to extend the abstract **`BaseFeature`** class. A requirement of the Java service loader mechanism is that plugins must provide a no-argument (no-arg) constructor. To help with this requirement, the **`BaseFeature`** class provides a

constructor that extensions can use to initialize their identification information as final values. It also provides the required methods for accessing the identification values.

The signature for the constructor in **`BaseFeature`** is shown in Code Listing 2.

Code Listing 2. Constructor for the `BaseFeature` Class

```
protected BaseFeature(String featureName, int majorVersion,
    int minorVersion, int patchVersion,
    boolean requiresConfiguration, String description);
```

Code Listing 3 shows an idiomatic no-arg constructor for a class that extends **`BaseFeature`**. Extensions of **`BaseHandler`** and **`BaseAgent`** follow the same pattern.

Code Listing 3. Idiomatic No-Arg Constructor for a Feature

```
class MyFeatureImpl extends BaseFeature implements MyFeature {
    public MyFeatureImpl() {
        super("MyFeature", 3, 1, 4, true, "This is my feature.");
      }
}
```

3.3.2 The Handler Interface

`Handler` is a subtype of **`Feature`** that has the ability to process **`Message`** objects. As stated previously, **`Message`** objects are used to communicate between **`Entity`** objects or between an **`Entity`** and the client application. The methods of the **`Handler`** interface are shown in Figure 8.

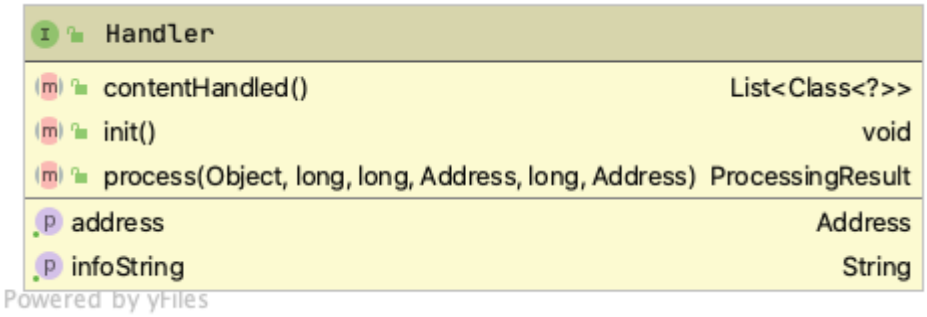

Figure 8. `Handler` interface

The **`Handler`** interface adds two methods to the **`Feature`** interface, **`contentHandled`** and **`process`**. Each **`Handler`** handles one or more message types and the **`contentHandled`**

method provides the message content classes that are handled by the **Handler**. This list is used by the message dispatch mechanism to route messages to their destination.

The **process** method causes a **Message** content object to be processed. The signature of the **process** method is shown in Code Listing 4.

Code Listing 4. Method Signature for the `Process` Method

```
ProcessingResult process(Object arg, long scheduledTime_ms,
    long currentTime_ms,  Address from,
    long id, Address respondTo);
```

The first argument is the content object delivered by the **Message**. The second argument, **scheduledTime_ms**, is used with scheduled events and specifies what time the **Event** was supposed to occur. The third argument is the current simulation time. Both times are provided to enable the **Handler** to adjust results if a scheduled event is overdue. **Events** and other **Message** types are discussed in Section 3.6. The fourth argument is the source of the message. The fifth argument is a unique sequence number for the message. The final argument is the **Address** to which the results of the **process** invocation should be sent. It is not necessary for the results to go back to the sender of the original message.

The result of a process call is a **ProcessingResult**, the details of which are shown in Figure 9. A **ProcessingResult** contains two public final fields. One is a list of new **Message** objects that were generated by the **Handler**. The **ProcessingResult** contains a list rather than just a single **Message** to enable a **Handler** to send out messages to multiple recipients. The second field contains the time for earliest **Event** in the list of messages. This value is used to enqueue the messages for processing. The **ProcessingResult** class also contains four static factory methods to simplify construction.

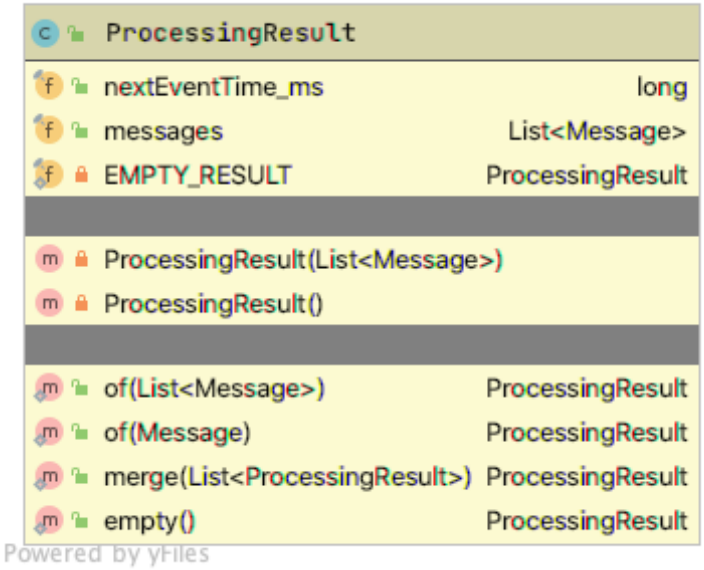

Figure 9. `ProcessingResult`

3.3.3 The `Agent` Interface

The **`Agent`** interface extends **`Handler`** and adds single method, **`tick`**. The **`tick`** method will be invoked on every on every simulation tick, that is, every iteration of the simulation event loop, regardless of whether or not the **`Agent`** has received a **`Message`**. The details of the **`Agent`** interface are shown in Figure 10.

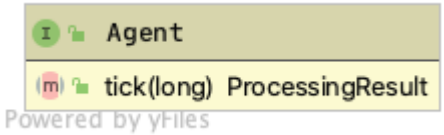

Figure 10. `Agent` interface

Like the process method in **`Handler`**, the **`tick`** method in **`Agent`** returns a **`ProcessingResult`**. Consequently, each tick can generate multiple messages that will be promulgated to the rest of the system. **`Agent`** will probably be the most commonly implemented **`Feature`** type because it offers the most flexibility.

3.3.4 `Feature` Specification

As stated in Section 3.3.1, instances of **`Feature`** are identified by the tuple of name, major, minor and patch version numbers, as well as their implementation class. When it is necessary to specify a **`Feature`** to be loaded, these parameters are encapsulated in the **`FeatureSpecification`** class, which is shown in Figure 11.

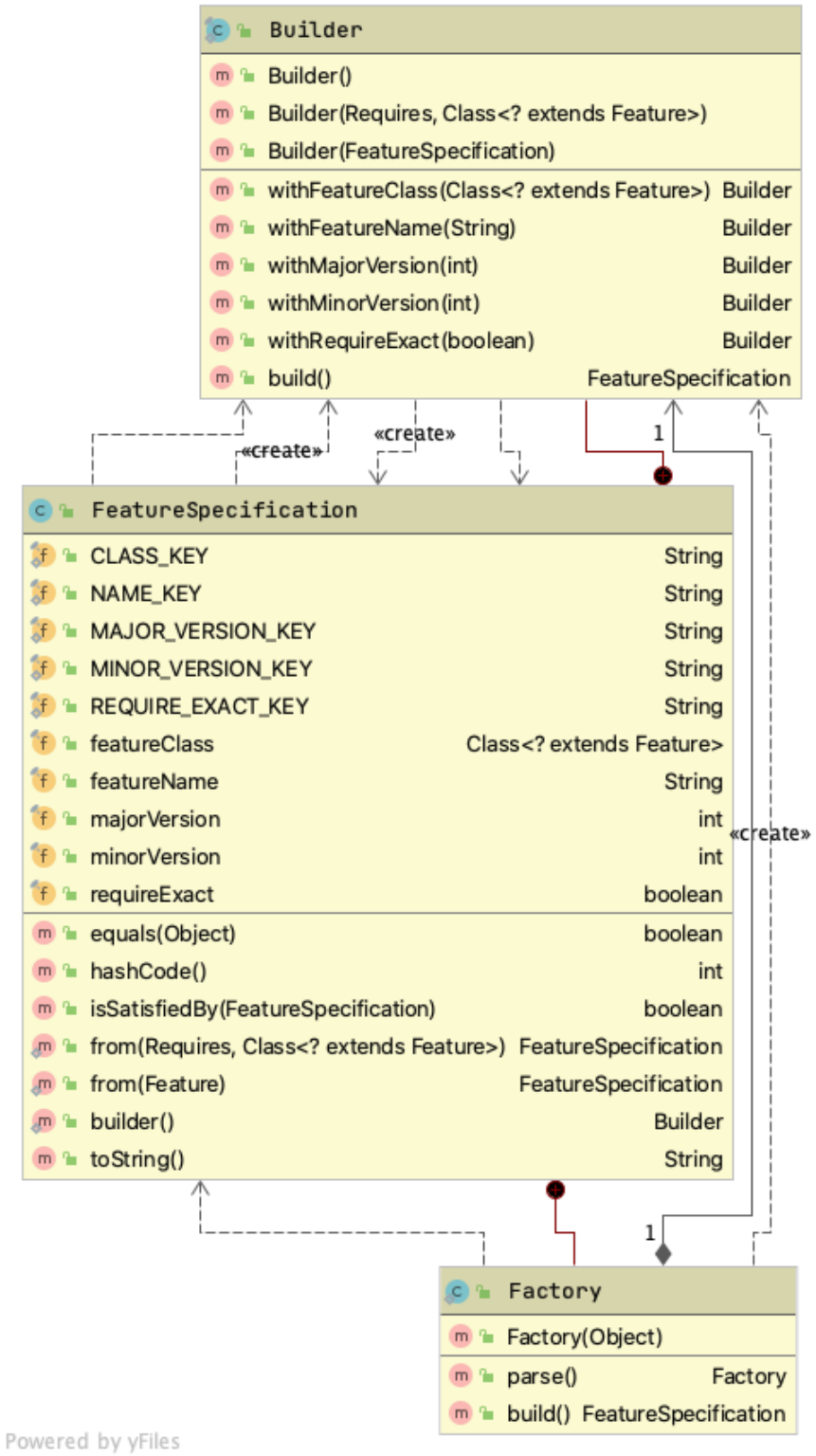

Figure 11. `Feature` specification

This approach of using names and versions to specify service gives SSTAF more flexibility than is normally found in Java service-based system. Typically, in a service-based system, the required service is identified simply by an interface or abstract class and the desired implementation is selected by placing the desired version on the classpath. The approach used by SSTAF enables different implementations of the same interface to be used concurrently within the system. The operation of this mechanism is discussed in Section 3.5.

One limitation of the approach is that all implementations must have a unique class name. This is because SSTAF currently does not create additional class loaders to partition the class namespace. If this limitation becomes a problem, future versions of SSTAF will address it.

The **`FeatureSpecification`** diagram also demonstrates one of the idioms used through SSTAF, specifically the use of inner **`Builder`** and **`Factory`** classes to assist in the construction of the class. This idiom is discussed in Section 3.4.1.1.

3.3.5 The `Requires` Annotation

The **`Requires`** annotation specifies an insertion point for a **`Feature`** in another **`Feature`**. Its contents are shown in Figure 12. When combined with the type of the field to which the annotation is attached, the fields in the annotation enable the generation of a **`FeatureSpecification`** that can be used for loading a **`Feature`** implementation. Both the requiring and required features can be of any **`Feature`** type (i.e., **`Feature`**, **`Handler`**, **`Agent`** or any subtype).

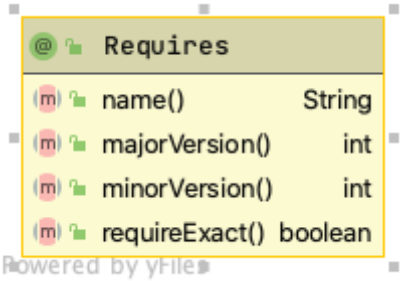

Figure 12. `Requires` annotation

Annotations support default values. The default value for the **`name`** field is the empty string. The default value for the **`majorVersion`** field is 1 and for the **`minorVersion`** field is 0. The **`requireExact`** field specifies whether candidate **`Feature`** must match the version number requirement exactly to be used or if a more recent version is acceptable. The default value for **`requireExact`** is false.

The following listings show some examples for using **`Requires`**.

Code Listing 5 uses all default values. The first **`Blackboard`** implementation found with a **`majorVersion`** greater than or equal to one will be assigned to the blackboard reference.

Code Listing 5. Example of `Requires` Annotation Use

```
public class MyAgent extends BaseAgent implements Awesome {
    @Requires
    Blackboard blackboard;
    //...
}
```

This example in Code Listing 6 sets name, **`majorVersion`** and **`minorVersion`**. The value of **`requireExact`** remains false, so the system will load the first **`PhysiologyModel`** implementation found with the name "Simple Physiology" and a version number greater than or equal to 3.7.0.

Code Listing 6. `Requires` Annotation with Minimum Version Specification

```
public class MyAgent extends BaseAgent implements Awesome {
   @Requires(name="Simple Physiology", majorVersion = 3,
       minorVersion = 7)
   PhysiologyModel physiology;
   //...
}
```

Code Listing 7 sets the **`requireExact`** flag to true, so only a **`Feature`** named "Simple Physiology" with a major version of 3 and a minor version of 7 will be accepted. Note that currently SSTAF does not use the patch-level value from the version number when matching requirements.

Code Listing 7. `Requires` Annotation with Exact Version Specification

```
public class MyAgent extends BaseAgent implements Awesome {

   @Requires(name="Simple Physiology", majorVersion = 3,
       minorVersion = 7, requireExact = true)
   PhysiologyModel physiology;
   //...
}
```

3.4 Entities

Within SSTAF, the primary items in the analysis are represented using the **`Entity`** class. This includes items such as humans, Soldiers and units. The **`Entity`** class hierarchy is shown in Figure 13. The **`Entity`** class itself is discussed in Section 3.4.1.

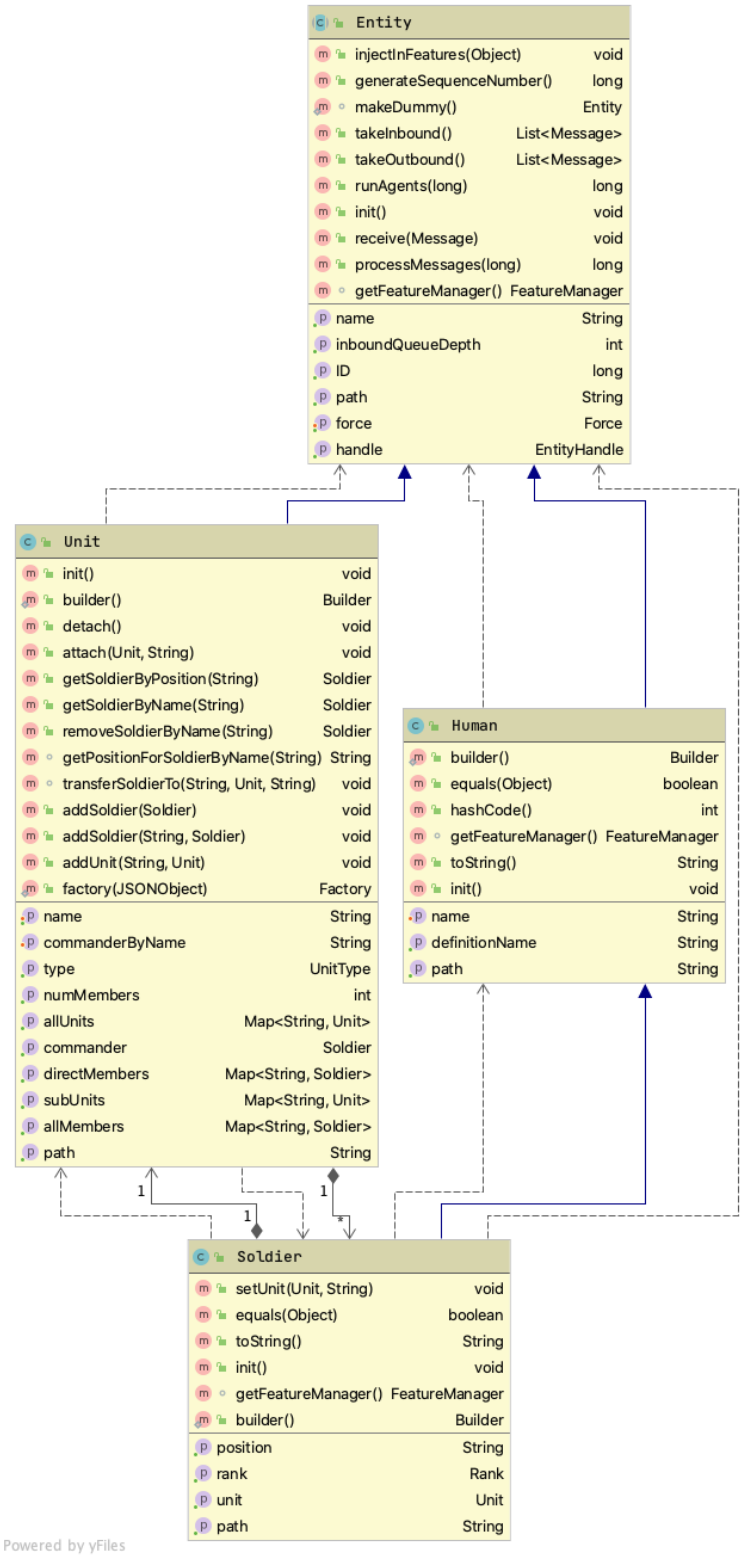

Figure 13. `Entity` class hierarchy

3.4.1 The `Entity` Class

At the base of the entity class hierarchy is the abstract **`Entity`** class, which is shown in Figure 14. The **`Entity`** class provides several capabilities that are central to how SSTAF works.

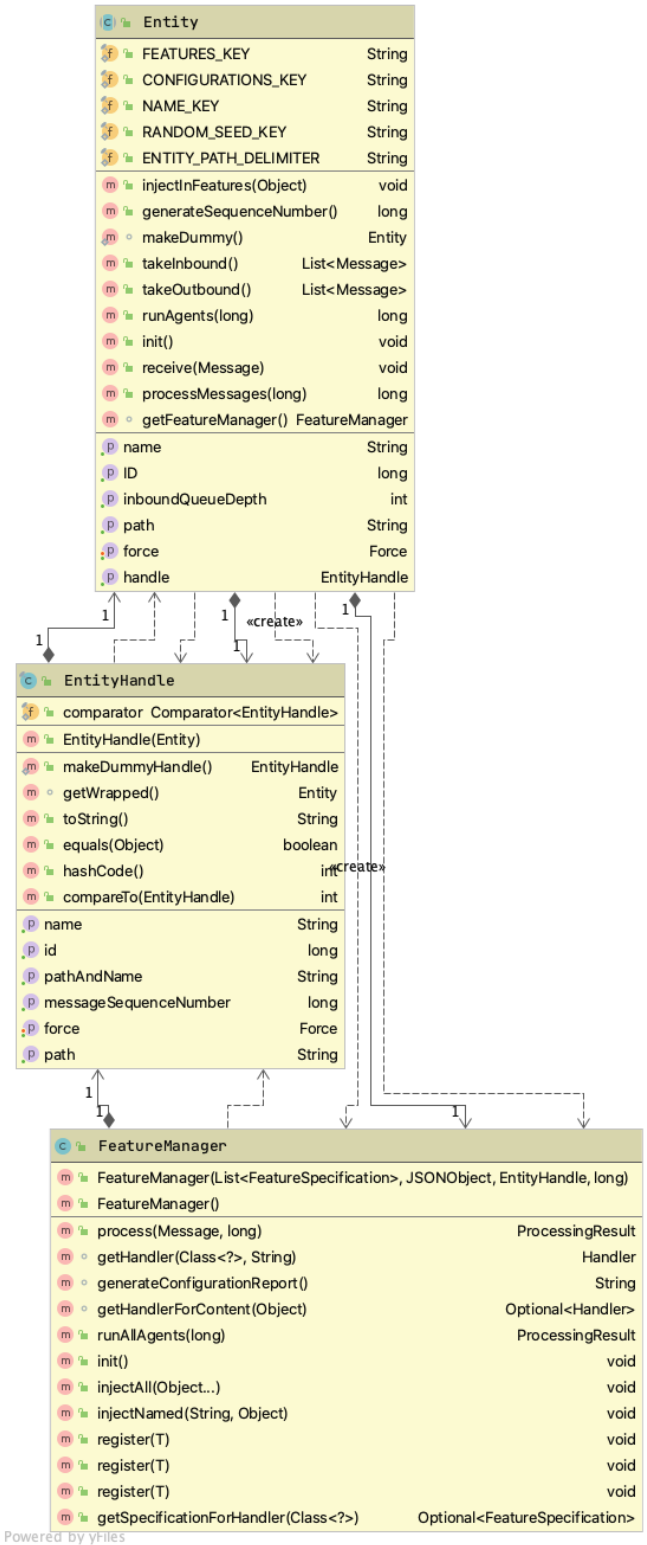

Figure 14. `Entity` class

Entity implements the **MessageDriven** interface. **MessageDriven** defines the API for objects that receive, process and emit messages. Figure 15 shows the **MessageDriven** interface. To support message-driven behavior, **Entity** includes both an inbound and an outbound queue. The inbound queue is a **PriorityQueue** and the outbound is a **ConcurrentLinkedQueue**. As is described in Section 3.9, the use of the **PriorityQueue** along with its associated **MessageComparator** is one of the mechanisms that help ensure repeatable analyses.

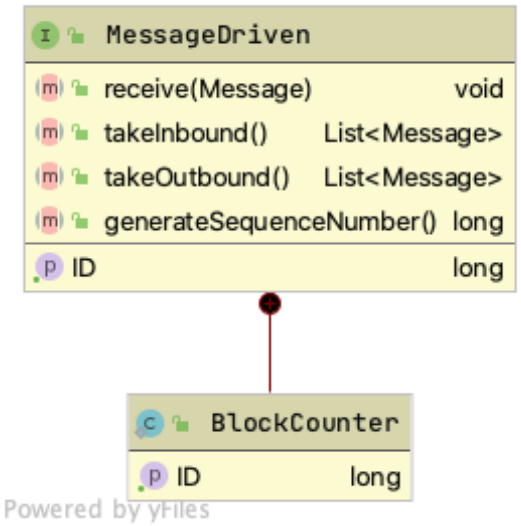

Figure 15. MessageDriven interface

The **Entity** class contains an instance of **FeatureManager**. **FeatureManager** is a utility class that is responsible for loading and holding features. It is also responsible for routing messages to the correct **Handler**, returning **Handler** results and propagating **tick** invocations from the event loop to the agents. The UML for the **FeatureManager** is shown in Figure 16.

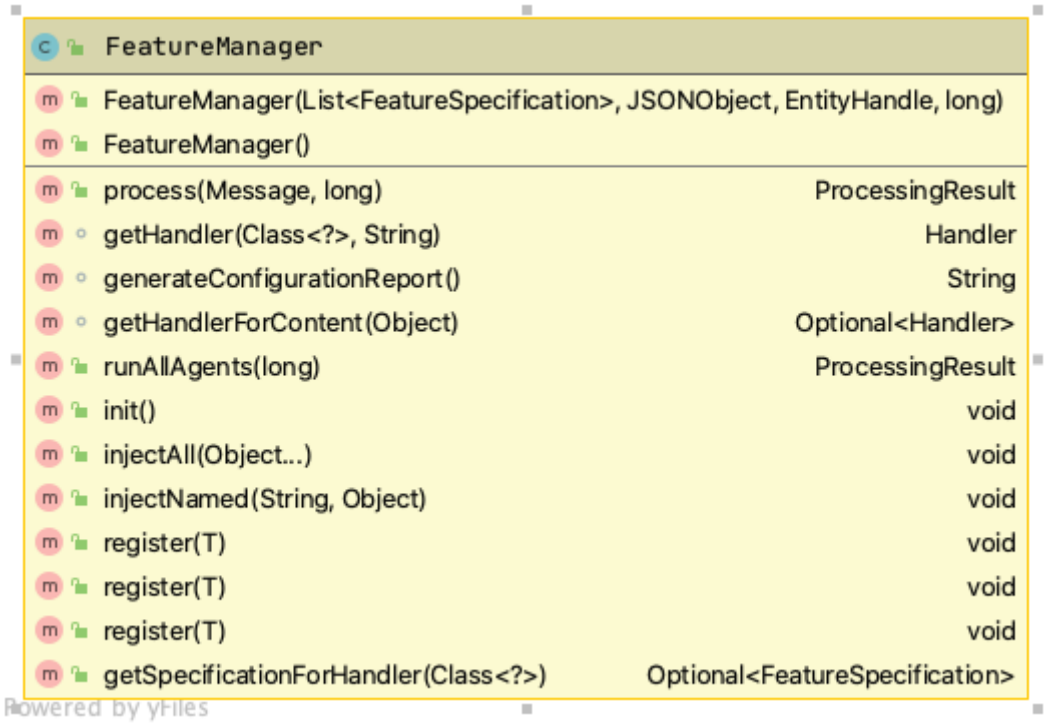

Figure 16. `FeatureManager` class

3.4.1.1 The `Factory` and `Builder` Inner Classes

One software idiom that is widely used in SSTAF is creating objects using a builder or factory method rather than using the **`new`** operator. The **`Entity`** class with its inner **`Factory`** and **`Builder`** classes demonstrates this idiom.

A *builder* is an object that is used to construct another object. A builder enables establishing the configuration for the object sequentially, usually through a series of setter methods. Once the configuration is complete, the object is instantiated by invoking a method on the builder (often **`build`**) that invokes the constructor for the desired object. When using the builder idiom, it is best practice to make the constructor for the class private to force the use of the builder.

There are several advantages to using builders rather than constructors. First, a builder can be configured incrementally; this eliminates the need to keep multiple temporary values to deliver to the constructor. The builder easily supports default and optional values. In Java, optional construction parameters often lead to an explosion of constructor methods. Using a builder with default values eliminates this problem. Another advantage of the builder is that, by convention, a setter in a builder returns the builder rather than **`void`**. This enables a more fluent style of code as is shown in Code Listing 8.

Code Listing 8. Fluent `Builder` Statement

```
SimpleMessage msg = SimpleMessage.builder().withDestination(d)
    .withRespondTo(this).withContents(stuff).build();
```

There are some complexities and limitations with builders in Java. In particular, extending a class that includes a builder can be tricky. Since the setter methods in the base class builder return the base class, one cannot arbitrarily mix base and derived class setters in a fluent configuration statement. This problem can be solved with some complex Java generics constructs but it is difficult to get correct. There are some classes in SSTAF 1.0 that still have issues with builder inheritance.

In SSTAF, **`Factory`** classes parse JSON input files and use builders to construct analysis domain objects. The factories instantiate builders and feed the configuration values from the JSON into the builders. The combination of the factories and the builders works well because the factories process the key-value pairs in the JSON incrementally.

3.4.1.2 Path

All **`Entity`** instances are identified by a unique path. The path is simply a colon-delimited string that starts at the top-level unit and proceeds down to the lowest unit. Within the lowest unit, Soldiers are identified by their position. The path mechanisms enable specific entities to be identified and accessed in a concise, human-readable way. A representative path is "3rdPlatoon:1stSquad:FireTeamAlpha:Grenadier".

3.4.1.3 `EntityHandle`

The **`EntityHandle`** class provides an indirect reference to an **`Entity`**. It also acts as a proxy for some of the entity's methods. **`EntityHandle`** objects, rather than direct **`Entity`** references, are used throughout SSTAF. This approach was taken in order to maintain strict control on **`Entity`**-to-**`Entity`** communication. To ensure repeatable analyses, it is necessary to control the order in which operations occurs. This would be very difficult if entities were able to make direct mutating calls to other entities in a concurrent environment. Using **`EntityHandle`** in the various APIs and interfaces helps ensure that communication is done through messages rather than through direct invocation. The details of the **`EntityHandle`** class are shown in Figure 17.

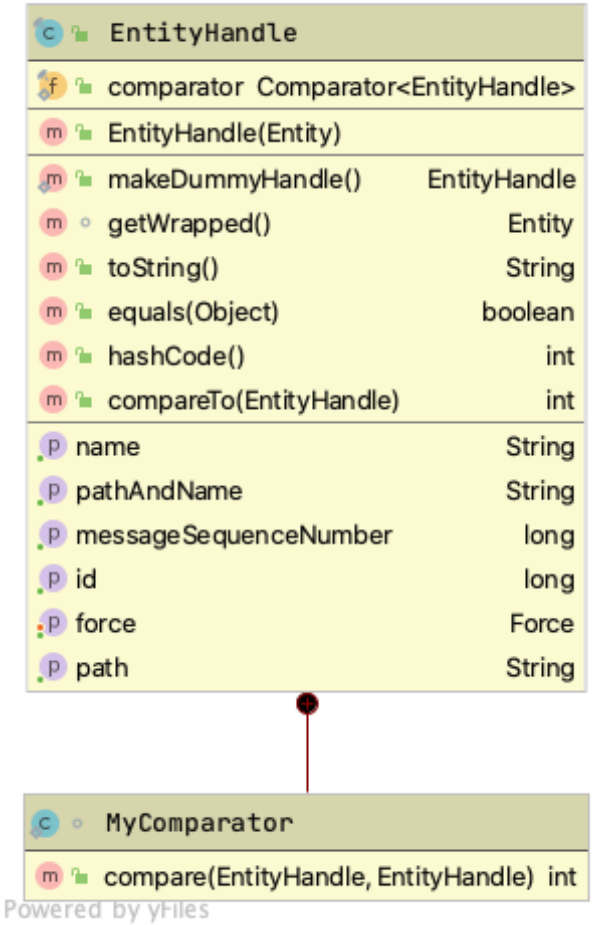

Figure 17. `EntityHandle` class

3.4.1.4 The `Injected` Annotation

Another small bit of infrastructure that helps with the configuration of the entities and features is the combination of the **`Injected`** annotation and the **`Injector`** utility class. Like the **`Requires`** annotation, these classes enable a simple dependency injection mechanism. However, **`Injected`** differs from **`Requires`** in two respects. First, **`Injected`** does not require the target to be a **`Feature`**; the injected object may be of any type. Second, **`Injected`** fields are not resolved automatically during construction. The owning object must use the methods in the **`Injector`** class to push values into the fields that have been annotated with **`Injected`**. For example, the **`Entity`** object used **`Injector`** to push its **`EntityHandle`** into the **`FeatureManager`**.

The **`Injected`** and **`Injector`** system acts as a setter for values that might or might not be required. This is cleaner than using setter methods, since the methods would have to be declared in the object interface. This would require the SSTAF framework to know the setter methods and thus would break the required independence of framework and features. The **`Injector`**

ignores unused injected values, so items can be injected eagerly without worrying about whether there is an annotated field to receive it.

3.4.2 Humans, Soldiers and Units

There are three existing sub-classes of **`Entity`** that are used to represent real-life items in SSTAF. These are the **`Human`**, **`Soldier`** and **`Unit`** classes.

The **`Human`** class is a thing wrapper over the **`Entity`** class. It adds one property, **`definitionName`**, which is the name associated with the configuration used to define the human. Multiple **`Human`** objects can be based on the same definition.

`Soldier` extends **`Human`** and adds several fields relevant to Soldiers, specifically rank, unit and position. The **`rank`** property holds the Soldier's rank. The **`Rank`** class is an enumeration of all Army ranks E1 to O10. The **`unit`** property is a reference to the unit to which this Soldier belongs. The position property is a **`String`** that specifies the role of the Soldier in the unit, for example *Commander*, *Grenadier* or *Clerk*.

The **`Unit`** class extends **`Entity`** directly. A unit contains a commander, organic Soldiers and optionally sub-units. The **`Unit`** class contains several methods for accessing and manipulating its member Soldiers and any sub-units.

3.5 Enhancing Entities with Features

At this point, all of the prerequisites have been presented and it is possible to discuss how entities are instantiated, features are added to them and dependencies between features are resolved. I illustrate the process by walking through a specific example of preparing a Soldier for use in the system.

To begin with, we have the JSON configuration file for the Soldier shown in Code Listing 9. The configuration file contains a single top-level JSON object that defines the Soldier. The **`JSONObject`** contains four fields, name, rank, features and configurations. As expected, the name and rank fields set the name and rank values for the Soldier. These values are optional. The features and configurations fields are required. However, with some restrictions either or both may be empty.

Code Listing 9. Soldier Configuration File

```
{
  "name": "Alvin York",
  "rank": "E5",
  "features": [
    {
      "featureName": "Dynamic Aim",
      "majorVersion": 0,
      "minorVersion": 1,
      "requireExact": "false"
    },
    {
      "featureName": "Soldier Task Agent",
      "majorVersion": 0,
      "minorVersion": 1,
      "requireExact": "false"
    },
    {
      "featureName": "Telemetry Agent",
      "majorVersion": 0,
      "minorVersion": 1,
      "requireExact": "false"
    }
  ],
  "configurations": {
    "Simple Anthropometry": "Anthropometry.json",
    "Kit Manager": "../common/StandardKit.json",
    "Dynamic Aim": "dynamicAimConfig.json",
    "Telemetry Agent": {
      "statesToRecord": [
        "Aim Internals",
        "Muscle Metrics"
      ]
    },
    "Soldier Task Agent": {
      "tickAim": true,
      "tickAim.range_m": 100,
      "tickAim.gun": "M4",
      "tickAim.mode": "STANDING"
    }
  }
}
```

The features array declares the features to be added to the entity. It contains a list of JSON objects, each of which will later be used to generate a **`FeatureSpecification`** object. Note that the specification blocks do not specify a **`Feature`** class. This is because all **`Features`** added to an **`Entity`** must be at least a **`Handler`** and are loaded as **`Handler`** instances. This is

because the only mechanism an **Entity** has to interact with a **Feature** is through messages and the **Handler** class is the lowest class in the **Feature** hierarchy that supports that capability.

In this example, the Soldier is declared to have three features: the Dynamic Aim model, the Soldier Task Agent and the Telemetry Agent. All three of these are demonstration features and are not ready for production use. They are discussed in Section 6.

The configurations object specifies the configurations to be provided to each of the features loaded into the Soldier. The configuration object mechanism leverages that JSON objects are key-value pairs. For feature configuration, the key is the name of the feature, as specified in the **FeatureSpecification** object, not its implementation class. The value of the configuration entry can be either a **JSONObject** with the configuration settings or a string that specifies a separate file that contains the settings.

In this example, five configuration objects are provided. There are the configurations for the three declared features. There are also two additional configurations. The first is for "Simple Anthropometry" and the second for "Kit Manager". Neither of these features was declared to be used, however; both are required by the "Dynamic Aim" model and are loaded automatically when that model is loaded.

As stated previously, the features and configurations sections can be empty under specific circumstances. If the feature block is empty, the configurations block may be empty. In this scenario, any content in the configurations block will be ignored since there are no features to which to apply the settings. If a feature is specified and that feature, or any of its dependencies, requires configuration and a configuration is not provided, an exception will be thrown.

To instantiate the Soldier, we use code such as that shown in Code Listing 10.

Code Listing 10. Soldier Construction Example

```
String filename = "input/soldiers/AlvinYork.json";
Soldier soldier = Soldier.Factory.from(filename, null)
    .parse().build();
```

A **Soldier** instance is constructed using the **Factory** and **Builder** classes mentioned earlier. The **from** method creates a new **Soldier.Factory** object from the given filename. The *null* value in the from invocation indicates that there is no parent **JSONObject** for the configuration. When the **JSONUtilities** class loads a **JSONObject** from a file, it embeds the filename and directory into the loaded object. Passing a parent object into a factory enables **JSONUtilities** to handle relative paths in file references as the system traverses input graphs. The factory in

turn instantiates a **`Soldier.Builder`**. The code for the **`Soldier.Factory`** and **`Soldier.Builder`** classes are shown in Code Listings 11 and 12.

Code Listing 11. `Soldier.Factory`

```
public static class Factory {
    private final JSONObject rootObject;
    private final Builder builder;

    private Factory(JSONObject definition) {
        rootObject = Objects.requireNonNull(definition,
            "JSONObject must not be null");
        builder = builder();
    }

    public static Factory from(final Object obj,
        final JSONObject parent) {

        if (obj instanceof JSONObject) {
            return new Factory(JSONUtilities.propagateFileInfo(
                (JSONObject) obj, parent));
        } else if (obj instanceof String) {
            JSONObject jsonObject = JSONUtilities.loadObjectFromFile(
                (String) obj, parent);
            return new Factory(jsonObject);
        } else {
            throw new SSTAFException("Value " + obj
                + " was not a String or JSONObject");
        }
    }

    public Factory parse() {
        new Human.Factory(rootObject, builder).parse();
        setOption(rootObject, RANK_KEY, String.class,
            builder::withRank);
        return this;
    }

    public Soldier build() {
        return builder.build();
    }
}
```

Code Listing 12. `Soldier.Builder`

```
public static class Builder extends Human.Builder {
    private Rank rank = Rank.E1;

    protected Builder() {
        super();
    }

    public void withRank(String rank) {
        this.rank = Rank.findMatch(rank);
    }

    public void withRank(Rank rank) {
        this.rank = Objects.requireNonNull(rank);
    }

    public Soldier build() {
        return new Soldier(this);
    }
}
```

There are two things to note in the parse method. First, the **`Soldier.Factory`** creates a **`Human.Factory`** and uses it to parse the configuration and put human-specific values into the **`Soldier.Builder`**. Second is the **`setOption`** method. The **`setOption`** method is a static method in **`JSONUtilities`** that reads a field of the specified type from the provided **`JSONObject`** and sends the read value to the method reference provided in the final argument. In this case, the method reference points to the **`withRank`** method in the **`Builder`**. The **`setOption`** method is used in this case because the rank value is optional. Similar methods in **`JSONUtilities`** are used to read mandatory values. These methods throw an exception if the required value is not found.

An excerpt from the **`Human.Factory`** class is shown in Code Listing 13. In its parse method, we see that it follows the same pattern as the **`Soldier.Factory`** class and invokes an **`EntityFactory`** to read entity-level values. After delegating to the **`EntityFactory`**, the **`Human.Factory`** reads the optional **`definitionName`** value.

Code Listing 13. **`Human.Factory`**

```
public static class Factory {
    //...
    public Factory parse() {
        new EntityFactory(rootObject, builder).parse();
        setOption(rootObject, CK_DEF_NAME, String.class,
            builder::withDefinitionName);
        return this;
    }
}
```

Code Listing 14 shows the parse method from the **`EntityFactory`**. In this method, we see that the name of the **`Entity`** is read if it exists. Additionally, a random number seed is read if it has been provided. SSTAF will use this seed to initialize, either directly or indirectly, all of the random number generators within this Entity. Random number management is discussed in Section 3.9.

Next, the **`JSONArray`** of features is read using the **`iterateOverMandatoryJSONObjects`** method. This method checks if the "features" tag exists and, if it does, iterates over all of the specifications declared within it. For each **`JSONObject`** within the list, it uses the **`FeatureSpecification.Factory`** to create a **`FeatureSpecification`** instance and add it to the builder. Finally, the factory reads the mandatory configurations field and passes that **`JSONObject`** to the builder's **`withConfigurations`** method.

Code Listing 14. `EntityFactory` Parse Method

```
public EntityFactory parse() {

    setOption(rootObject, NAME_KEY , String.class, builder::withName);

    JSONUtilities.setOption( rootObject, RANDOM_SEED_KEY, Long.class,
        builder::withRandomSeed);

    iterateOverMandatoryJSONObjects( rootObject, FEATURES_KEY,
        spec -> {
            FeatureSpecification featureSpecification =
                new FeatureSpecification.Factory(spec)
                    .parse().build();
            builder.withSpecification(featureSpecification);
    });

    setMandatoryOrThrow(rootObject, CONFIGURATIONS_KEY,
        JSONObject.class, builder::withConfigurations);

    return this;
}
```

The builder for the base **`Entity`** is shown in Code Listing 15. The builder for the **`Human`** class extends this builder, and the builder for Soldier extends **`Human.Builder`**. In this class, we see that the features to add are held as a list of specifications and the configurations are retained as the original **`JSONObject`**. The build method is abstract because **`Entity`** itself is an abstract class.

Code Listing 15. Entity Builder

```
public abstract static class Builder {

    protected Long id = BlockCounter.userCounter.getID();
    protected String name = UUID.randomUUID().toString();
    protected Long randomSeed = System.currentTimeMillis();

    protected List<FeatureSpecification> specifications =
        new ArrayList<>();
    protected JSONObject configurations = new JSONObject();

    public void withSpecification(FeatureSpecification spec) {
        specifications.add(Objects.requireNonNull(spec,
            "Service specification must not be null"));
    }

    public void withConfigurations(final JSONObject configurations) {
        this.configurations = Objects.requireNonNull(configurations,
            "Feature configurations must not be null");
    }

    public void withRandomSeed(final long randomSeed) {
        this.randomSeed = randomSeed;
    }

    public void withName(String val) {
        this.name = Objects.requireNonNull(val,
            "Name must not be null");
    }

    public abstract Entity build();
}
```

At this point, the **`Soldier`** can be constructed using the build method. Referring back to the **`Soldier.Builder`**, we see that the build method simply calls the **`Soldier`** constructor and provides itself as the only argument. The three relevant constructors, **`Soldier`**, **`Human`** and **`Entity`** are shown in Code Listings 16, 17, and 18.

Code Listing 16. `Soldier` Constructor

```
protected Soldier(Builder builder) {
    super(builder);
    this.rank = Objects.requireNonNull(builder.rank);
}
```

Code Listing 17. `Human` Constructor

```
protected Human(Builder builder) {
    super(builder);
    this.definitionName = builder.definitionName;
    this.uuid = builder.uuid;
}
```

Code Listing 18. `Entity` Constructor

```
protected Entity(Builder builder) {
    logger.trace("Constructing Entity from builder {}",
        builder::toString);

    myID = builder.id;
    handle = new EntityHandle(this);
    randomGenerator = new MersenneTwister();
    randomSeed = builder.randomSeed == null ? myID :
        builder.randomSeed;
    this.name = builder.name;

    featureManager = new FeatureManager(builder.specifications,
        builder.configurations, handle,
        RNGUtilities.generateSubSeed(randomGenerator));
    logger.debug("Entity '{}' constructed, features = {}", name,
        featureManager.generateConfigurationReport());
}
```

As can be seen, the **`Soldier`** and **`Human`** constructors invoke their superclass constructor and then they set their field values. Two interesting things happen in the **`Entity`** constructor. First, the constructor creates a random number generator. The generator is a **`MersenneTwister`** provided by the Apache Commons Math library. The random number generator is then seeded using the value from the configuration file if one was provided, or, if a seed was not provided, the entity identification number (**`myID`**). Entity identification numbers are unique and repeatable, so random number seeds will be consistent from run to run even if a seed is not provided. The second interesting action is the creation of a **`FeatureManager`**, which, in turn, creates a **`Resolver`**. This is where loading features and resolving dependencies occurs.

Code Listing 19 shows the constructor for the **`FeatureManager`**. The **`FeatureManager`** sets its owner to be an **`EntityHandle`** to its containing an **`Entity`**. It then creates a new random number generator using the provided seed. Next, the constructor creates a **`Resolver`**. This is the class that does all of the heavy lifting for managing dependencies. Finally, **`FeatureManager`** iterates over the list of **`FeatureSpecifications`** that the configuration file defined and uses the **`Resolver`** to load each **`Feature`** and any transitive dependencies.

Code Listing 19. `FeatureManager` Constructor

```
public FeatureManager(List<FeatureSpecification> features, JSONObject
configurations, EntityHandle owner, long randomSeed) {

    this.owner = owner;
    RandomGenerator generator = new MersenneTwister(randomSeed);
    Resolver resolver = new Resolver(this.features, configurations,
        owner, RNGUtilities._generateSubSeed_(generator));

    features.forEach(spec -> {
        Feature f = resolver.loadAndResolveDependencies(spec);
        if (f instanceof Agent) {
            register((Agent) f);
        } else if (f instanceof Handler) {
            register((Handler) f);
        } else {
            register(f);
        }
    });
}
```

Code Listing 20 shows the **`loadAndResolveDependencies`** method for the **`Resolver`** class. This method begins by first checking to see if the required **`Feature`** has already been loaded by querying the feature cache. The feature cache is a map of **`FeatureSpecifications`** to **`Features`** held by the **`FeatureManager`**. If the **`Resolver`** does not find a match, it uses the **`ServiceLoader`** mechanism to load a **`Feature`** matching the specification. If that operation is successful, **`loadAndResolveDependencies`** adds the loaded **`Feature`** to the cache and then passes it to the **`resolveDependencies`** method. Otherwise, the method throws an exception.

Code Listing 20. Resolver `LoadAndResolveDependencies` Method

```
public Feature
    loadAndResolveDependencies(FeatureSpecification specification) {
    logger.debug("{} - Loading and resolving for {}",
        owner.getPath(), specification);
    Feature feature = findMatchInCache(specification, featureCache);
    if (feature == null) {
        logger.trace("{} - Feature not in cache, loading {}",
            owner.getPath(), specification);
        Optional<Feature> optionalFeature = _load_(Feature.class,
            specification);
        if (optionalFeature.isPresent()) {
            feature = optionalFeature.get();
            featureCache.put(FeatureSpecification._from_(feature),
                feature);
            logger.debug("{} - Feature loaded and added to cache.  {}"
,
                 owner.getPath(), featureCache.keySet());
            resolveDependencies(feature);
        } else {
            logger.error("Could not load service {}",
                specification.toString());
            Loaders._printAvailableServices_(Feature.class);
            throw new SSTAFException("Could not load service '"
                + specification.toString() + "'");
        }
    }
    return feature;
}
```

Code Listing 21 shows the **`resolveDependencies`** method. For the specified **`Feature`**, this method invokes **`loadRequiredServices`** to find and load all dependencies. It then caches all of the loaded dependencies and finally applies the configuration object to the **`Feature`**.

Code Listing 21. Resolver `resolveDependencies` Method

```
public void resolveDependencies(Feature feature) {
    logger.trace("Resolving dependencies for {}",
        feature.getClass().getName());
    var rv = loadRequiredServices(feature, featureCache);
    featureCache.putAll(rv);
    //...   Logging
    Optional<JSONObject> optConfig = getConfiguration(
        feature.getName());
    optConfig.ifPresent(configuration ->
        feature.configure(configuration,
            RNGUtilities.generateSubSeed(generator)));
}
```

The most complex part of the **`Feature`** resolution system is the **`loadRequiredServices`** method shown in Code Listing 22. Given an initial **`Feature`** and a **`FeatureCache`**, this method recursively follows **`Feature`** dependencies specified by the **`Requires`** annotations. The method begins by creating its own cache that is a copy of the parent cache. This is necessary because Java maps do not allow modification in recursive methods. Next, Java reflection is used to find all fields that are annotated with the **`Requires`** tag. If **`loadRequiredServices`** finds one, it checks the field to see if a value is assigned to it. If so, it is skipped. If not, a **`FeatureSpecification`** is generated from the **`Requires`** annotation and the field type. The specification is then checked against the cache. If a match is found, it is used. If no match is found, the **`Loaders`** class is used to load the required **`Feature`**. If that fails, the method throws an exception. If it succeeds, the newly loaded **`Feature`** is added to the cache and the method then calls itself, using the newly loaded **`Feature`** as the source for new requirements. Once a **`Feature`** is loaded and its dependencies resolved, it is configured using the configuration object from the input if one was provided.

Code Listing 22. The `loadRequiredServices` Method

```
private Map<FeatureSpecification, Feature>
    loadRequiredServices(Feature target,
        Map<FeatureSpecification, Feature> parentCache) {
    logger.trace("Loading required services for {}",
        target.getName());
    Map<FeatureSpecification, Feature> myCache =
        new HashMap<>(parentCache);

    for (Field field : getFieldsWithRequires(target)) {
        logger.trace("{} - Processing Requires field {}",
            target.getName(), field.getName());
        Requires requires = field.getAnnotation(Requires.class);
        Class<?> rawClass = field.getType();

        if (isFieldSet(target, field)) {
            logger.trace(
                "{} - Skipping field {} because it is already set",
                target.getName(), field.getName());
        } else {
            Class<? extends Feature> clazz =
                rawClass.asSubclass(Feature.class);
            FeatureSpecification spec =
                FeatureSpecification.from(requires, clazz);

            Feature toInject = findMatchInCache(spec, myCache);
            if (toInject != null) {
                logger.debug("{}/{} Found match in cache, reusing {}",
                    target.getName(), field.getName(),
                    spec.toString());
            } else {
                logger.debug("{}/{} Getting new instantiation of {}",
                        target.getName(), field.getName(),
                        spec.toString());
                final Feature newlyLoaded =
                        Loaders.loadAsRef(clazz, spec.featureName,
                            spec.majorVersion, spec.minorVersion,
                            spec.requireExact);
                if (newlyLoaded == null) {
                    String identifier =
                        spec.featureName == null ||
                        spec.featureName.length() == 0
                        ? spec.featureClass.getName()
                        : spec.featureName;
                    throw new SSTAFException(
                        "Could not load an implementation for "
```

```
                        + identifier);
                }

                //
                // Since this is a new load, recurse to fill it in.
                //
                FeatureSpecification fs =
                    FeatureSpecification.from(newlyLoaded);
                myCache.put(fs, newlyLoaded);
                logger.trace(
                    "{} - Registered {} in cache, {} entries",
                    target.getName(), fs.toString(), myCache.size());
                logger.trace("{} - Recursing now for {}",
                    target.getName(), newlyLoaded.getName());
                var childCache = loadRequiredServices(newlyLoaded,
                    myCache);
                myCache.putAll(childCache);
                logger.trace("{} - done with {}, {} entries",
                    target.getName(), newlyLoaded.getName(),
                    myCache.size());

                //
                // Inject the ownerHandle into the feature.
                // This must be done before configuration
                // or an NPE might occur.  Some feature loggers
                // reference owner paths.
                //
                logger.trace("{} - Injecting owner handle {}",
                        target.getName(), owner.getPath());
                Injector._inject_(newlyLoaded, owner);

                Optional<JSONObject> optConfig =
                    getConfiguration(newlyLoaded.getName());
                optConfig.ifPresentOrElse(
                    configuration -> {
                    logger.trace("{} - Configuring {} with {}",
                        target.getName(), newlyLoaded.getName(),
                        configuration.toString(2));
                    newlyLoaded.configure(configuration,
                        RNGUtilities.generateSubSeed(generator));
                    },
                    () -> logger.info("{} - No configuration for {}",
                            target.getName(), newlyLoaded.getName())
                );
                toInject = newlyLoaded;
            }
            //
            // Inject the service into the field.
            //
            logger.debug("{} - Injecting {} into field {}",
                target.getName(), toInject.getName(),
```

```
                field.getName());
            Injector._injectField_(target, field, toInject);
        }
    }
    return myCache;
}
```

Once the **FeatureManager** construction has completed, all of the requirements for the **Entity** have been loaded and configured.

3.6 Messages and Addresses

As stated previously, a **Message** is an object that is used to invoke a command or request information from an **Entity**, Also, the **Message** instance itself is just an envelope. The important part is the message content. That said, there is a small **Message** class hierarchy. Figure 18 shows the **Message** hierarchy.

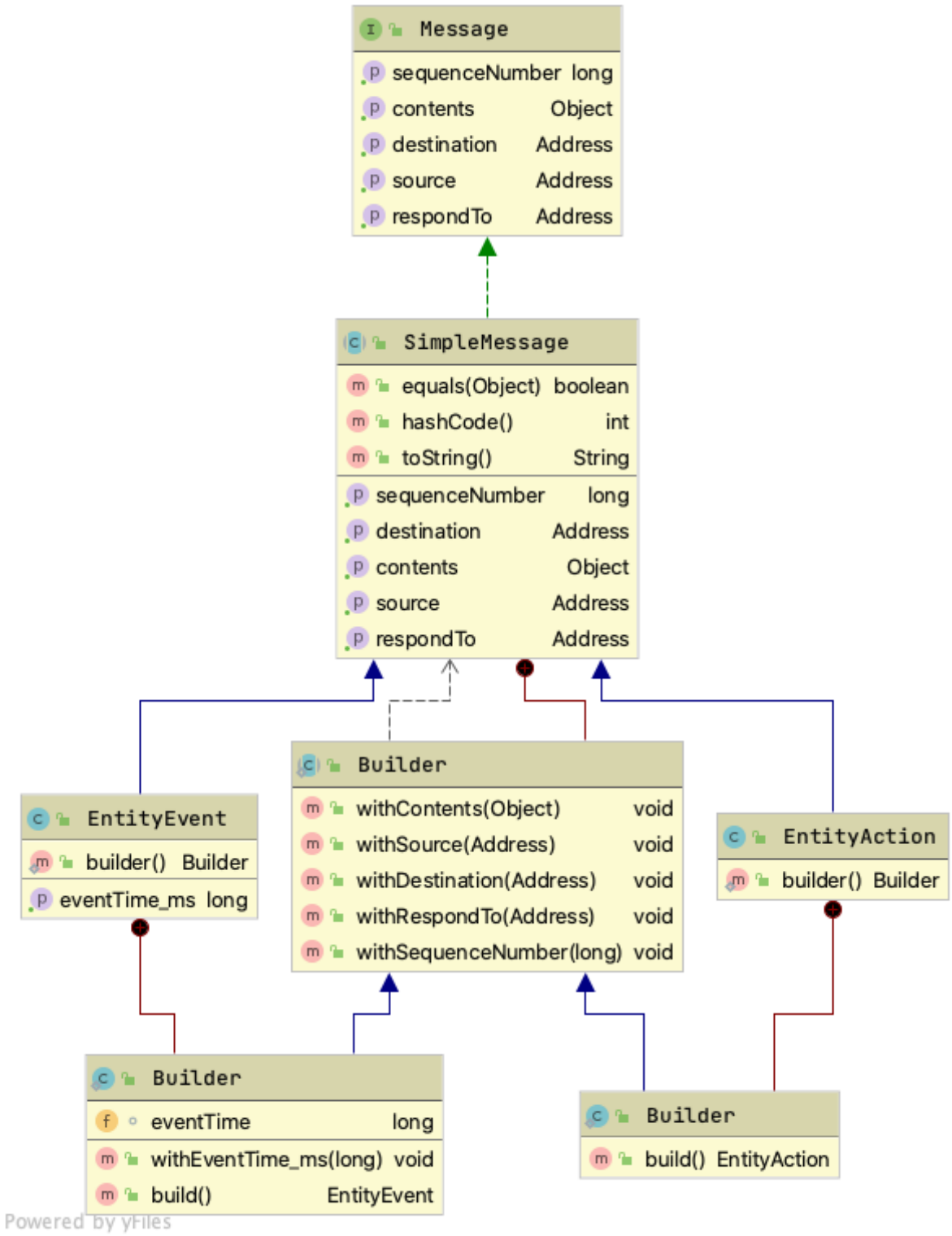

Figure 18. `Message` hierarchy

At the base of the hierarchy is the **`Message`** interface. It provides the methods necessary for routing the message, uniquely identifying it and retrieving the contents.

The **sequenceNumber** method returns the message number generated by the source **Entity**. Each **Entity** has its own message counter, so the combination of **Entity** and **sequenceNumber** is a unique identifier for the **Message**. The **contents** method returns the object that is the payload of the message. The three remaining methods, **destination**, **source** and **respondTo,** each return an **Address** object. The **Address** class is exactly what its name implies, a specific location in the system. In **Message,** the addresses specify the source of the message, the recipient of the message and the destination for any message resulting from the processing of the original message. The **respondTo** location need not be the same as the source address.

The **Address** class consists of two values, an **EntityHandle** and a **String**, which specify the name of a **Handler**. If a destination **Address** includes a **Handler** name, the **FeatureManager** routes the message to that specified **Handler**, regardless if it is the registered **Handler** for the content type.

SimpleMessage is an abstract implementation of **Message** that includes all of the necessary fields and accessors. It also includes a builder to help with construction.

There are two concrete implementations of **SimpleMessage**. The first, **EntityAction**, specifies messages that are to be processed immediately by the recipient. Once the **EntityAction** is delivered, the **Entity** will process it the next time the **processMessages** method is invoked. The second class, **EntityEvent**, includes an additional field to specify the time of the event. This class is used to schedule an action that will be processed in the future. The content of an **EntityEvent** is processed when the simulation clock meets or exceeds the specified **eventTime_ms** value.

3.7 Session and Session Messages

The **Session** class provides the surface API between client applications and the SSTAF system. Recall from the discussion on requirements that one requirement was to have a simple and stable API. As shown in Figure 19, the **Session** class is indeed simple. It contains only three significant methods: **submit**, **tick** and **asyncTick.**

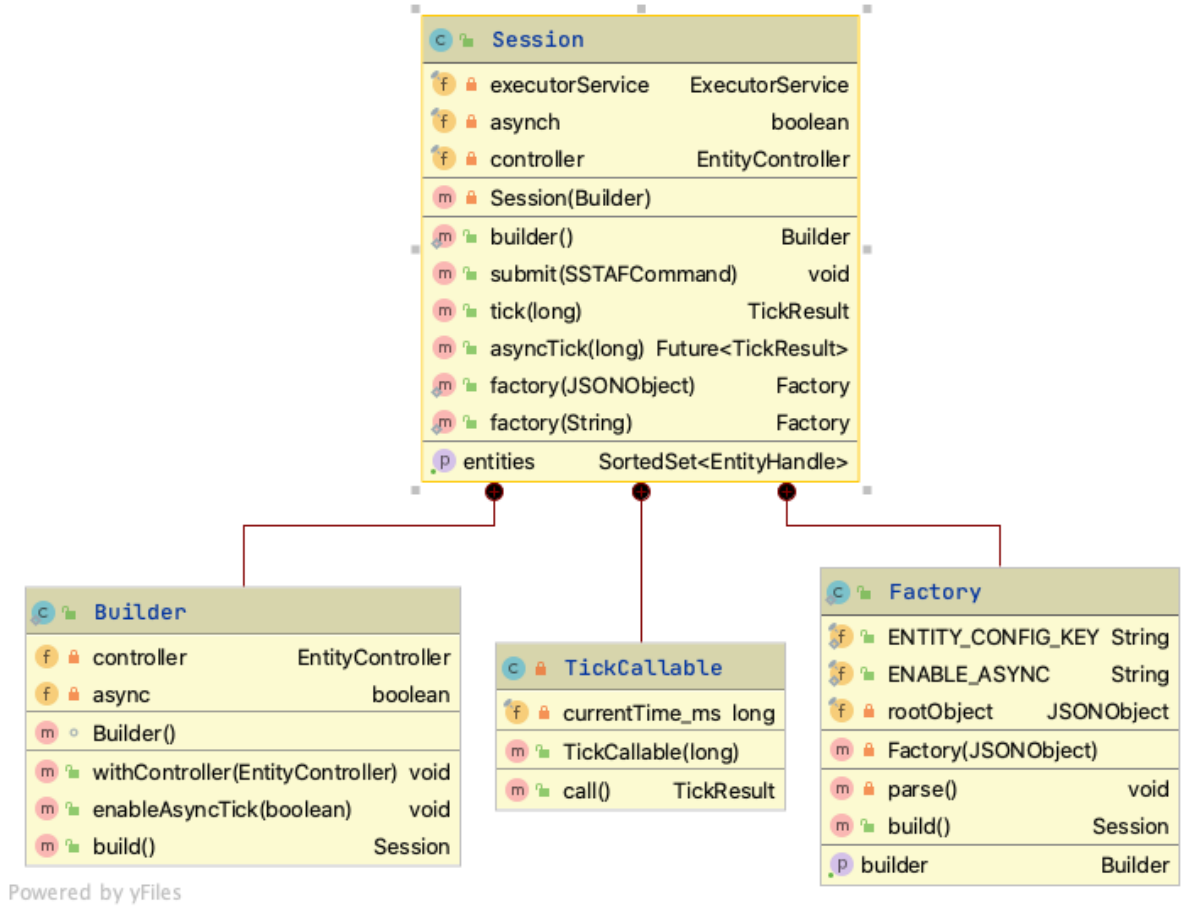

Figure 19. `Session` class and its inner classes

The **`submit`** command pushes a **`SSTAFCommand`** message into the **`EntityController`** for routing. **`Session`** use a different set of message classes than that used between entities (see Section 3.5). Using different message types helps ensure a firewall between the client application and SSTAF internals.

Figure 20 shows the **`SSTAFCommand`** class hierarchy. The hierarchy consists of two classes, **`SSTAFCommand`** and **`SSTAFEvent`**. These classes parallel the **`EntityAction`** and **`EntityEvent`** message classes. **`SSTAFCommand`** and **`SSTAFEvent`** differ from their entity peers in that rather than using an **`Address`** object to specify the recipient, the SSTAF messages use an **`EntityHandle`**. This simplification is reasonable since the application should not override the default **`Handler`** mappings by specifying a **`Handler`** name. **`SSTAFCommand`** also eliminates the **`source`** and **`respondTo`** fields, since all **`SSTAFCommand`** messages originate from and return to the client application. Like the **`EntityEvent`**, the **`SSTAFEvent`** adds the event time.

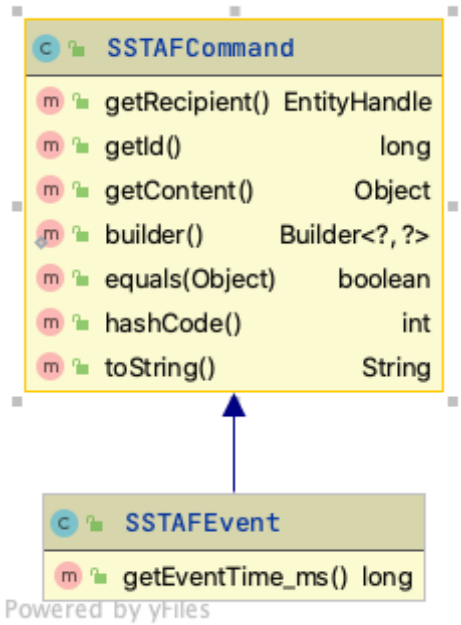

Figure 20. `SSTAFCommand` and `SSTAFEvent`

In SSTAF, a tick is the process of advancing the simulation clock. SSTAF does not control the time step of a tick. The client application controls the time step. Therefore, all tick methods include an argument for the new simulation time. The **`Session`** can process a tick either synchronously or asynchronously. The synchronous mode is invoked with the **`tick`** method, while the asynchronous mode is invoked using the **`asyncTick`** method. In synchronous mode, **`Session`** invokes the **`tick`** method of the **`EntityController`** directly, the main thread blocks and a **`TickResult`** object is returned, eventually. In asynchronous mode, the **`Session`** creates a **`TickCallable`** object to wrap the call to the **`EntityController.`** The **`TickCallable`** object is dispatched to a separate thread. The **`asyncTick`** method returns a **`Future`** for the **`TickResult`** immediately. This enables the main thread to perform additional work while the SSTAF threads do their job.

Figure 21 shows the **`TickResult`** returned from the tick. It contains two values, a list of **`SSTAFResult`** objects and the time of the next event that is queued within system.

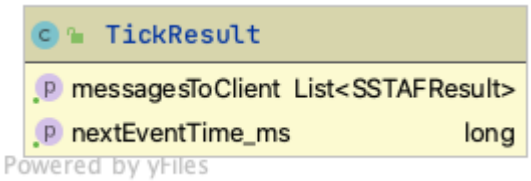

Figure 21. `TickResult` class

The **`SSTAFResult`** hierarchy is separate from the **`SSTAFCommand`** and entity message hierarchies. The two **`SSTAFResult`** types are shown in Figure 22. The **`SSTAFResult`** contains the successful result of a previously issued **`SSTAFCommand`** or **`SSTAFEvent`**. The result contains the ID of the original message. If something failed during execution, a **`SSTAFError`** is returned.

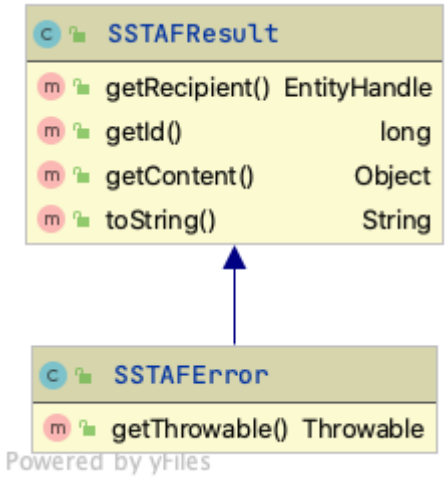

Figure 22. `SSTAFResult` and `SSTAFError`

3.8 `EntityController`

The **`EntityController`**, shown in Figure 23, is the master controller for the SSTAF system. It has many responsibilities, including the following:

- loading entities,
- routing messages between entities,
- processing simulation ticks, and
- routing messages to and from the client application.

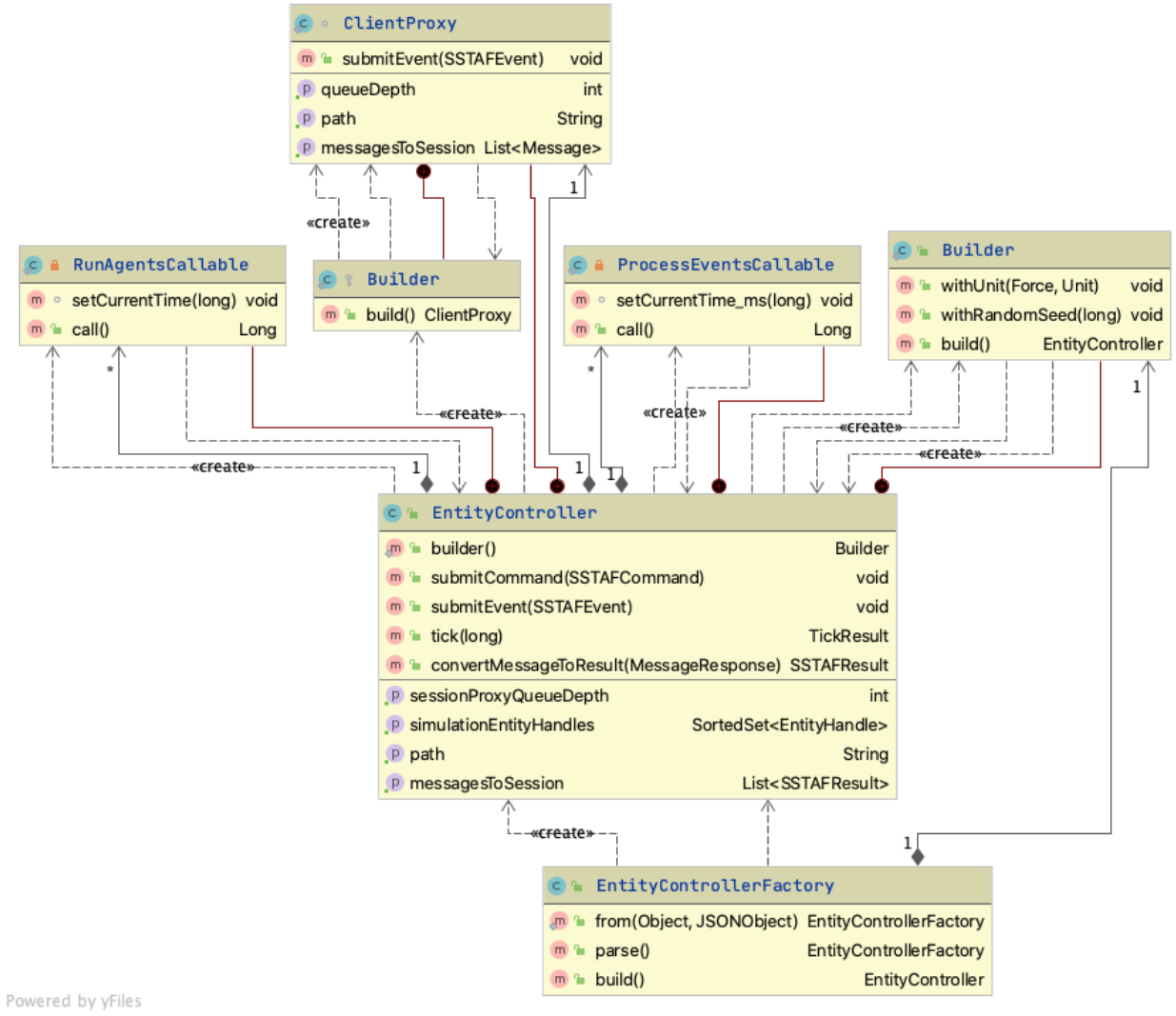

Figure 23. `EntityController` class and its inner classes

An important aspect of the **`EntityController`** is that it is also an **`Entity`** itself. This means that it can be extended with features. An important difference between features loaded into a regular entity as opposed to those loaded into the **`EntityController`** is that the **`EntityController`** injects its features with the **`EntityRegistry`**. The **`EntityRegistry`** is an object that contains references to all of the entities in the system. This enables **`EntityController`** features to access every **`Entity`** and act as coordination agents for features in the individual entities. Section 4.3 provides an example of an **`Agent`** for the **`EntityController`** that acts as a coordinator.

The central method in the SSTAF system is the **`tick`** method in **`EntityController`**. This method is responsible for driving all of the activity within entities and their features. Because of

the importance and the method and some of its complexities, I walk through it in detail. Code Listing 23 shows the **`tick`** method in its entirety.

Code Listing 23. The `tick` Method

```
public TickResult tick(long currentTime_ms) {
    logger.debug("Executing tick at {}", currentTime_ms);

    List<Future<Long>> nextTimes1;
    try {
        runAgentsTasks.forEach(task ->
            task.setCurrentTime(currentTime_ms));
        nextTimes1 = executorService.invokeAll(runAgentsTasks);
    } catch (InterruptedException e) {
        e.printStackTrace();
        nextTimes1 = List.of();
    }
    routeMessages();

    this.processMessages(currentTime_ms);
    this.runAgents(currentTime_ms);
    routeMessages();

    List<Future<Long>> nextTimes2;
    try {
        processEventsTasks.forEach(task ->
            task.setCurrentTime_ms(currentTime_ms));
        nextTimes2 = executorService.invokeAll(processEventsTasks);
    } catch (InterruptedException e) {
        e.printStackTrace();
        nextTimes2 = List.of();
    }
    routeMessages();

    List<SSTAFResult> toSession = getMessagesToSession();
    long nextEventTime_ms = Long.min(getMinTime(nextTimes1),
        getMinTime(nextTimes2));
    return new TickResult(nextEventTime_ms, toSession);
}
```

The **`tick`** method begins by invoking the agents in the analysis entities. Starting with the first **`try`** block, **`tick`** iterates over the **`runAgentTasks`** list and sets the current simulation time in each task. The next step is to submit the list of tasks to an **`ExecutorService`** that uses a pool of threads to run all of the tasks.

To maximize performance, the **`EntityController`** uses multiple threads to perform processing within the individual entities. The **`runAgentsTask`** list contains an instance of the

RunAgentCallable class for each **Entity** in the system. The **RunAgentCallable** is an implementation of **java.util.concurrent.Callable** that wraps the invocation of one entity's **runAgents** method in its **call** method. The result of the **runAgents** call is the time for next simulation event created by agents for the **Entity**. The **ExecutorService** returns a list of **Future** objects to those results.

The next step is to route any messages generated by the agents to the appropriate destination. The **routeMessages** method takes messages from the **outboundQueue** in each entity and moves it to the **inboundQueue** in the destination entity.

At this point, **tick** switches to invoking the features in the **EntityController**. First, any messages and executable events are processed. Next, all of the agents in the **EntityController** are activated and then any new messages are routed. Note that this section of **tick** is single-threaded.

In the second **try** block, the messages and events for each entity are processed identically to how the entity agents were dispatched. The **processEventsTask** list contains **ProcessEventsCallable** objects that wrap the **processMessages** method in the **Entity**.

Again, the method routes messages to the appropriate destinations. The **getMessagesToSession** method gathers the messages that are destined for the **Session** and converts outbound messages from their internal classes to the appropriate **Session** message class.

The next step is to determine the next event time by computing the minimum from both time lists. Finally, the method creates a **TickResult** object and returns it to the **Session**.

3.9 Repeatability

The ability to repeat a run and generate consistent results is important for analyses. I designed SSTAF to produce repeatable results even when **Session** is using multiple threads to process **Entity** actions and events. I accomplished this using several techniques including careful management of random number generators and draw order.

First, each **Entity** has its own random number generator that is seeded from a single parent generator. The parent generator is seeded from the session configuration file. Instantiation of the entities happens in the same order each time, because the entity graph is defined in the input files and entities are instantiated by a single thread when the session is loaded. Thus, each **Entity** starts with a repeatable, and independent, random state.

Next, communication between entities is only through messages and messages have a deterministic and repeatable execution order. The repeatable execution order is enforced by the

PriorityQueue and **MessageComparator** mentioned in Section 3.4.1. This mechanism ensures that on each tick, the messages will be executed in the same order for each run. Since the messages will be executed in the same order, the state changes within an entity will be executed in the same order and thus the state of the **Entity** after every tick will be repeatable.

Finally, SSTAF makes direct access of one **Entity** by another difficult. Direct access would enable one **Entity** to manipulate the state of another outside of the message-passing mechanism. This could change the order of operations and break repeatability. This is especially true if the **Session** is using multiple threads.

3.10 Verification

SSTAF is not a model. SSTAF does not simulate anything unless the system has been augmented with user-specified models. Therefore, it is not necessary to validate SSTAF. However, it is a complex software system, so verification of its functional correctness is imperative.

To ensure functional correctness, multiple layers of testing are used. These layers include unit, module and integration tests. SSTAF uses the Junit testing framework and follows the Java convention of placing the unit and module tests within the **src/test** directory within each module. Integration tests are contained within the **mil.sstaf.integration** module.

4. IMPLEMENTING FEATURES

As stated previously, the primary mechanism for adding capability to SSTAF is through the addition of features to entities. In this section, I discuss the design, implementation and testing of features using a series of examples:

1. **`Blackboard`**
   Provides a cache that other **`Features`** within an **`Entity`** can use. The entries on the **`Blackboard`** can be set to expire at a specific simulation time.
2. **`TelemetryAgent`**
   Provides the ability to log values from the **`Blackboard`**.
3. **`MobilityEntityAgent`**
   Provides the ability for an entity to move on a two-dimensional surface.
4. **`MobilityCentralAgent`**
   Tracks the current position of all entities in the simulation. This agent is an **`EntityController`** feature.

4.1 Blackboard

The **`Blackboard`** addresses the need for the features within an entity to be able to post values to a central location so that other features can read them. Initially, this capability was built directly into the **`Entity`** class; however, later it was refactored as a **`Feature`**.

4.1.1 Requirements

The following are the requirements for the **`Blackboard`** feature:

- Enable transient storage of arbitrary objects using a **`String`** as an identification key.
- Enable retrieval of an object that is associated with a **`String`** key.
- Enable removal of an object that is associated with a **`String`** key.
- Enable each stored object to have a time range for which it is valid.
- Expired objects are eligible for removal.
- Enable stored objects to be perpetually valid.
- Enable storage action to be invoked either directly or through a message.
- Enable retrieval action to be invoked either directly or through a message.
- Enable removal action to be invoked either directly or through a message.

4.1.2 Build Configuration

Although not required, it is advisable to use test-driven development (TDD) when implementing features for SSTAF. In order to do TDD, it is first necessary to have a minimal structure in place so that one can build the code and execute the tests. Once one has the minimal structure in place,

developing the feature is simply a matter of developing tests to check requirements and writing code to pass the tests.

As stated previously, SSTAF features are implemented as Java modules. The idiomatic approach for SSTAF is to divide the code for the feature into two modules. The first module specifies the API for the feature. If the feature has a custom interface, that interface class is included in the API module. Additionally, any custom classes that are used in the interface or as message content must be included in the API module. The API module is a compile-time dependency for any SSTAF-based component that uses that feature. The second module contains the implementation for the feature. It depends on the API module and must be available on the module path at runtime.

Given this idiom, we create two directory hierarchies for the **`Blackboard`** using the Gradle-standard "src/main/java/…, src/test/java/…" pattern. The API module is under **`mil.sstaf.blackboard.api`**. Because the implementation is to be solely in memory, the implementation module is called **`mil.sstaf.blackboard.inmem.`**

The **`build.gradle`** file for the API module is simple and is shown in Code Listing 24. It shows that the API depends only on the SSTAF core module, the JSON module, and the Apache Commons Math module.

Code Listing 24. The `build.gradle` file for the `Blackboard` API Module

```
ext.moduleName = 'mil.sstaf.blackboard.api'
dependencies {
    implementation project(':framework:mil.sstaf.core')
    compile "org.json:json:20190722"
    compile group: 'org.apache.commons', name: 'commons-math3',
        version: '3.6.1'
}
```

Similarly, the **`build.gradle`** for the implementation module is also simple (Code Listing 25).

Code Listing 25. The `build.gradle` file for the `Blackboard` Implementation Module

```
ext.moduleName = 'mil.sstaf.blackboard.inmem'
dependencies {
    implementation project(':framework:mil.sstaf.core')
    implementation project(':features:support:mil.sstaf.blackboard.api
')
}
```

At this point, it is possible to start developing tests and code.

4.1.3 `Blackboard` Interface

Given that the requirements specified in Section 4.1.1 specify that the **`Blackboard`** must support interaction through messages, the appropriate type of **`Feature`** to implement is the **`Handler`**. The new **`Blackboard`** interface thus extends the **`Handler`** interface. This gives us the class inheritance hierarchy shown in Figure 24.

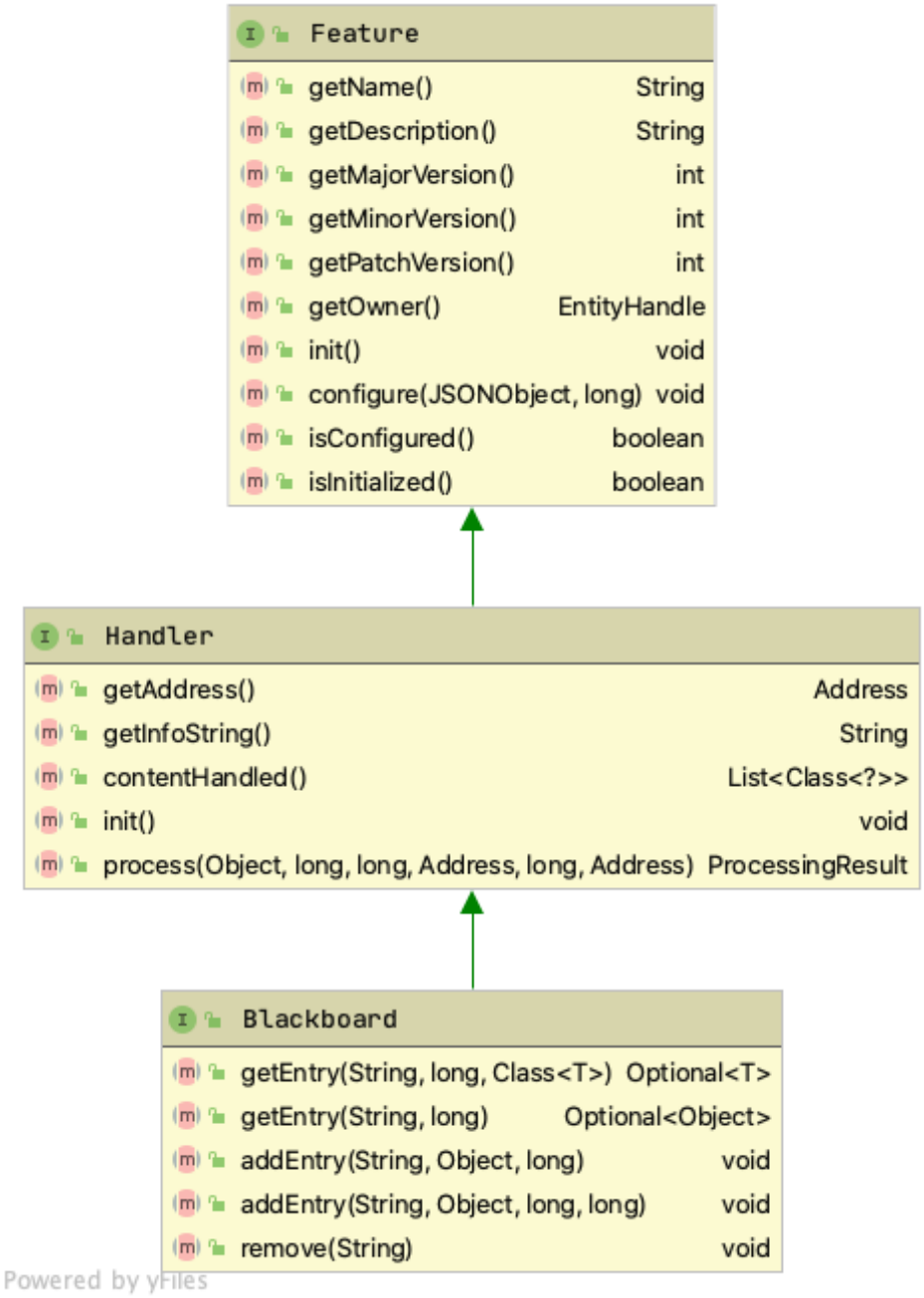

Figure 24. Inheritance hierarchy for the `Blackboard` interface

To support the requirements, the **`Blackboard`** interface includes five methods as shown in Code Listing 26. The first two support retrieval of an object. The two **`addEntry`** methods register an object with the **`Blackboard`**. The intent of the first version is that the object is available beginning at **`timestamp_ms`** and never expires. In the second version, the object is available at **`timestamp_ms`** and expires at **`expiration_ms`**. The two **`getEntry`** methods enable retrieval from the **`Blackboard`**. In both methods, the entry key and current simulation

time are provided to locate the value and determine if it is valid. In the first form, the stored object is returned as simple **`Object`**. In the second form, a match is checked against the specified type and returned as that type. In both methods, the **`Optional`** class is used to wrap the return value rather than returning a **`null`** if no match is found. The final method, **`remove`**, deletes the specified object from the **`Blackboard`**.

Code Listing 26. The `Blackboard` Interface

```
public interface Blackboard extends Handler {

    void addEntry(final String key, final Object value,
        final long timestamp_ms);

    void addEntry(final String key, final Object value,
        final long timestamp_ms, final long expiration_ms);

    Optional<Object> getEntry(final String key,
        final long currentTime_ms);

    <T> Optional<T> getEntry(final String key,
        final long currentTime_ms, Class<T> type);

    void remove(final String key);
}
```

4.1.4 Message Classes

To enable the **`Blackboard`** to accept commands through messages, it is necessary to create the messages. To convey the commands, there are three classes: **`AddEntryRequest`**, **`GetEntryRequest`** and **`RemoveEntryRequest`**. Similarly, there are three class to convey the results of the commands: **`AddEntryResponse`**, **`GetEntryResponse`** and **`RemoveEntryResponse`**. The contents of these classes are shown in Figures 25 through 30.

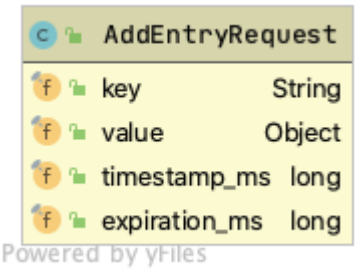

Figure 25. `AddEntryRequest` class

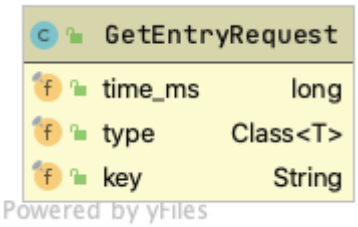

Figure 26. `GetEntryRequest` class

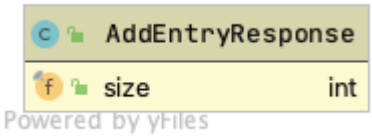

Figure 27. `AddEntryResponse` class

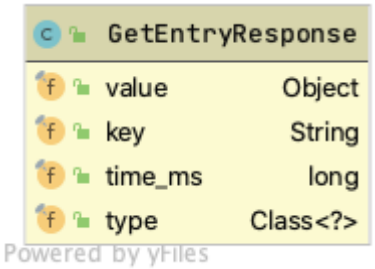

Figure 28. `GetEntryResponse` class

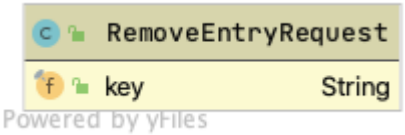

Figure 29. `RemoveEntryRequest` class

Figure 30. `RemoveEntryResponse` class

4.1.5 Implementation Class

At this point, it is possible to present the implementation of the **`Blackboard`** interface in the **`InMemBlackboard`** class. Because the class must implement **`Handler`**, it is appropriate to extend **`BaseHandler`**. This gives the class inheritance hierarchy shown in Figure 31.

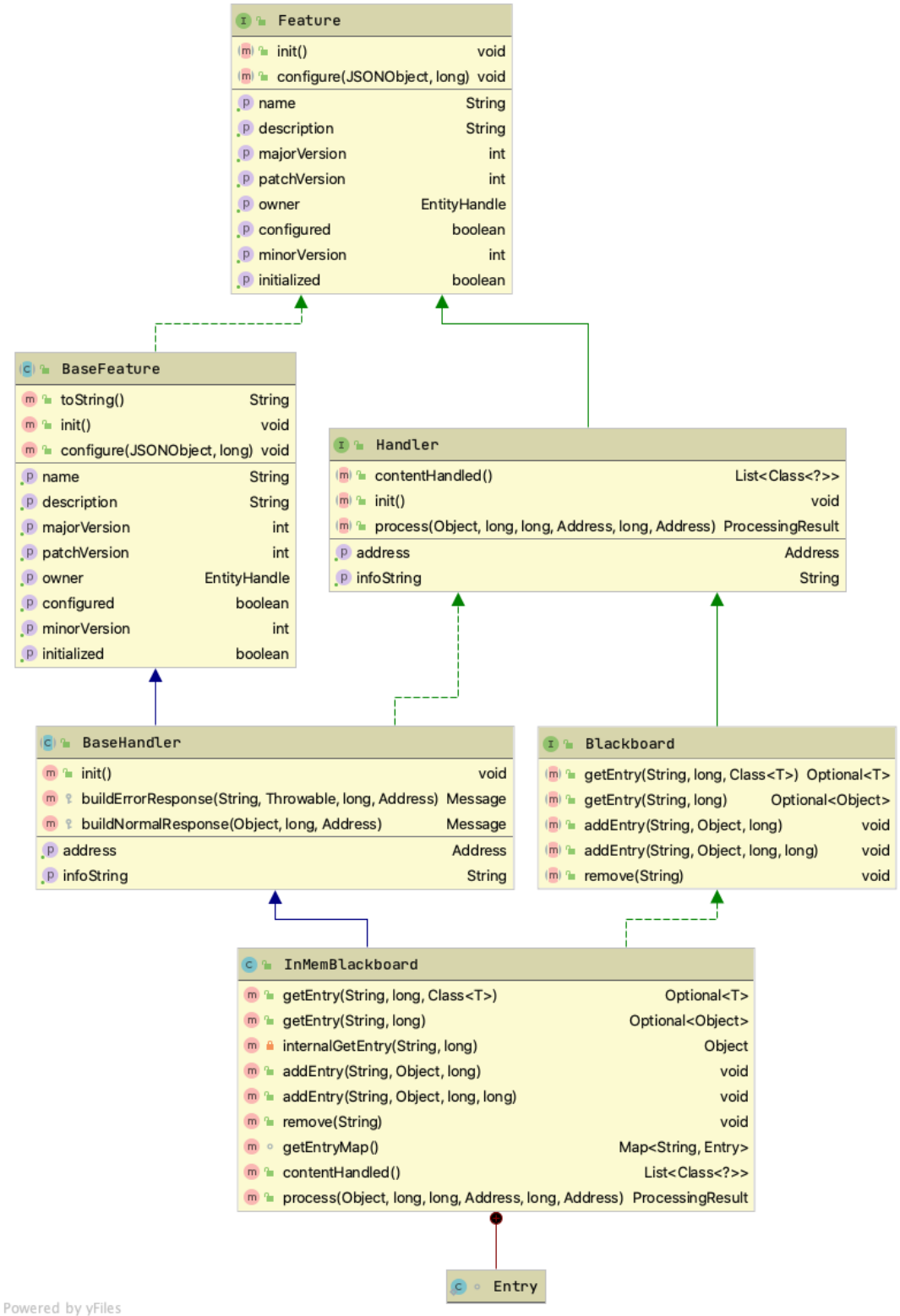

Figure 31. The `InMemBlackboard` class hierarchy

From the diagram, we can see that in addition to the methods specified in the **`Blackboard`** interface, the **`InMemBlackboard`** must implement the **`contentHandled`** and **`process`** methods. Code Listing 27 shows the **`contentHandled`** method. Code Listing 28 shows the **`process`** method.

Code Listing 27. The `ContentHandled` Method

```
public List<Class<?>> contentHandled() {
    return List.of(EntityHandle.class, AddEntryRequest.class,
        GetEntryRequest.class, RemoveEntryRequest.class);
}
```

Code Listing 28. The `Process` Method

```
public ProcessingResult process(Object arg, long scheduledTime_ms,
    long currentTime_ms, Address from, long id, Address respondTo) {
    Message output;
    if (arg instanceof GetEntryRequest) {
        GetEntryRequest<?> message = (GetEntryRequest<?>) arg;
        Object value = internalGetEntry(message.key, message.time_ms);
        GetEntryResponse response = new GetEntryResponse(value,
            message.key, message.time_ms, message.type);
        output = buildNormalResponse(response, id, respondTo);
    } else if (arg instanceof RemoveEntryRequest) {
        RemoveEntryRequest rer = (RemoveEntryRequest) arg;
        remove(rer.key);
        RemoveEntryResponse response =
            new RemoveEntryResponse(entryMap.size());
        output = buildNormalResponse(response, id, respondTo);
    } else if (arg instanceof AddEntryRequest) {
        AddEntryRequest aer = (AddEntryRequest) arg;
        addEntry(aer.key, aer.value, aer.timestamp_ms,
            aer.expiration_ms);
        AddEntryResponse response =
            new AddEntryResponse(entryMap.size());
        output = buildNormalResponse(response, id, respondTo);
    } else if (arg instanceof EntityHandle) {
        EntityHandle entityHandle = (EntityHandle) arg;
        addEntry(entityHandle.getPath(), entityHandle, BIGBANG,
            FOREVER);
        AddEntryResponse response =
            new AddEntryResponse(entryMap.size());
        output = buildNormalResponse(response, id, respondTo);
    } else {
        output = this.buildErrorResponse("Message class '"
            + arg.getClass() + "' is not supported.",
            new IllegalArgumentException(), id, respondTo);
    }
    return ProcessingResult.of(output);
}
```

Note that in addition to the three request classes, **`InMemBlackboard`** also accepts an **`EntityHandle`** as a message. As shown in the **`process`** method, if an **`EntityHandle`** is received, it is added to the **`Blackboard`** with the entity path as the key. It is also always valid,

with its valid period ranging from **BIGBANG** to **FOREVER**. In retrospect, this was a mistake and it will be removed in the next version. It is more appropriate for the caller to use the **AddEntryRequest** message rather than add this special case for a single scenario.

To support the storage and retrieval of entries, **InMemBlackboard** uses a standard **HashMap**. Each submitted object is wrapped in an **Entry** class that also contains the beginning and ending times for the validity of the object. Code Listing 29 shows the **addEntry** method.

Code Listing 29. The `AddEntry` Method

```
public void addEntry(final String key, final Object value,
    final long timestamp_ms, final long expiration_ms) {
    Objects.requireNonNull(key, "Blackboard entry may not have a null
key");
    Objects.requireNonNull(value,
        "Blackboard entry may not have a null value");
    if (logger.isDebugEnabled()) {
        logger.debug("Adding '{}':'{}'; valid {} to  {}",
                key, value, timestamp_ms, expiration_ms);
    }
    entryMap.put(key, new Entry(value, timestamp_ms, expiration_ms));
}
```

The **InMemBlackboard** class does not require configuration nor does it require initialization. Therefore, it is sufficient to use the implementations for the **configure** and **init** methods provided in the base classes.

4.1.6 Module Configuration

The **module_info.java** file for the API module is shown in Code Listing 30. It shows that the API module requires only the **mil.sstaf.core** module. Additionally, the module exports the **mil.sstaf.blackboard.api** package to make it publicly available.

Code Listing 30. The `Module_info.java` File for the Blackboard API

```
module mil.sstaf.blackboard.api {
    exports mil.sstaf.blackboard.api;
    requires transitive mil.sstaf.core;
}
```

The **module_info.java** file for the implementation module (Code Listing 31) is more interesting. It shows that the implementation requires the **Blackboard API** and **JSON** modules and that it exports the **mil.sstaf.blackboard.inmem** package. The **provides** and **opens**

statements support service loading. The **`provides`** statements indicate that the module provides specific services with an implementations of the specified class. In this case, the **`Feature`**, **`Handler`** and **`Blackboard`** capabilities are all provided by the **`InMemBlackboard`** class. The **`opens`** statement enables the **`mil.sstaf`**.core module to use the Java reflection mechanism to interact with the **`mil.sstaf.blackboard.inmem`** module. This access is required in order to do the dependency injection provided by the **`Requires`** and **`Injected`** mechanisms.

Code Listing 31. `Module_info.java` File for the Blackboard Implementation Module

```
import mil.sstaf.blackboard.api.Blackboard;
import mil.sstaf.blackboard.inmem.InMemBlackboard;
import mil.sstaf.core.features.Feature;
import mil.sstaf.core.features.Handler;

module mil.sstaf.blackboard.inmem {
    exports mil.sstaf.blackboard.inmem;

    requires mil.sstaf.blackboard.api;
    requires org.json;

    provides Feature with InMemBlackboard;
    provides Handler with InMemBlackboard;
    provides Blackboard with InMemBlackboard;

    opens mil.sstaf.blackboard.inmem to mil.sstaf.core;
}
```

4.2 `TelemetryAgent`

During development, I discovered that it would be very useful to be able to read the values in the **`Blackboard`** and record them in a structured form. The **`TelemetryAgent`** reads values from its entity's **`Blackboard`** and writes the values to a file.

4.2.1 Requirements

The requirements for the **`TelemetryAgent`** are the following:

- Enable the user to specify which values for each **`Entity`** are recorded.
- Record the telemetry for each value in a separate file.
- Place all telemetry files for an **`Entity`** in a separate directory.
- Build a directory hierarchy based on **`Entity`** paths.
- Record telemetry on each tick.
- Record telemetry in a comma-separated value (CSV) file.
- Automatically generate column headers.

4.2.2 The `StateProperty` Annotation

The **`StateProperty`** annotation is used to mark fields in an object so that they can be written by the **`TelemetryAgent`**. **`StateProperty`** contains one parameter, **`headerLabel`**, the value of which is used to for the column header in the output file. If the value is not set, the name of the field is used as the column header.

4.2.3 Implementation

Because the **`TelemetryAgent`** does not provide capabilities beyond automatically recording data, it is not necessary to develop a new interface. It is sufficient to implement it as a simple extension of **`BaseAgent`**. Figure 32 shows an abbreviated version of the inheritance hierarchy for **`TelemetryAgent`**.

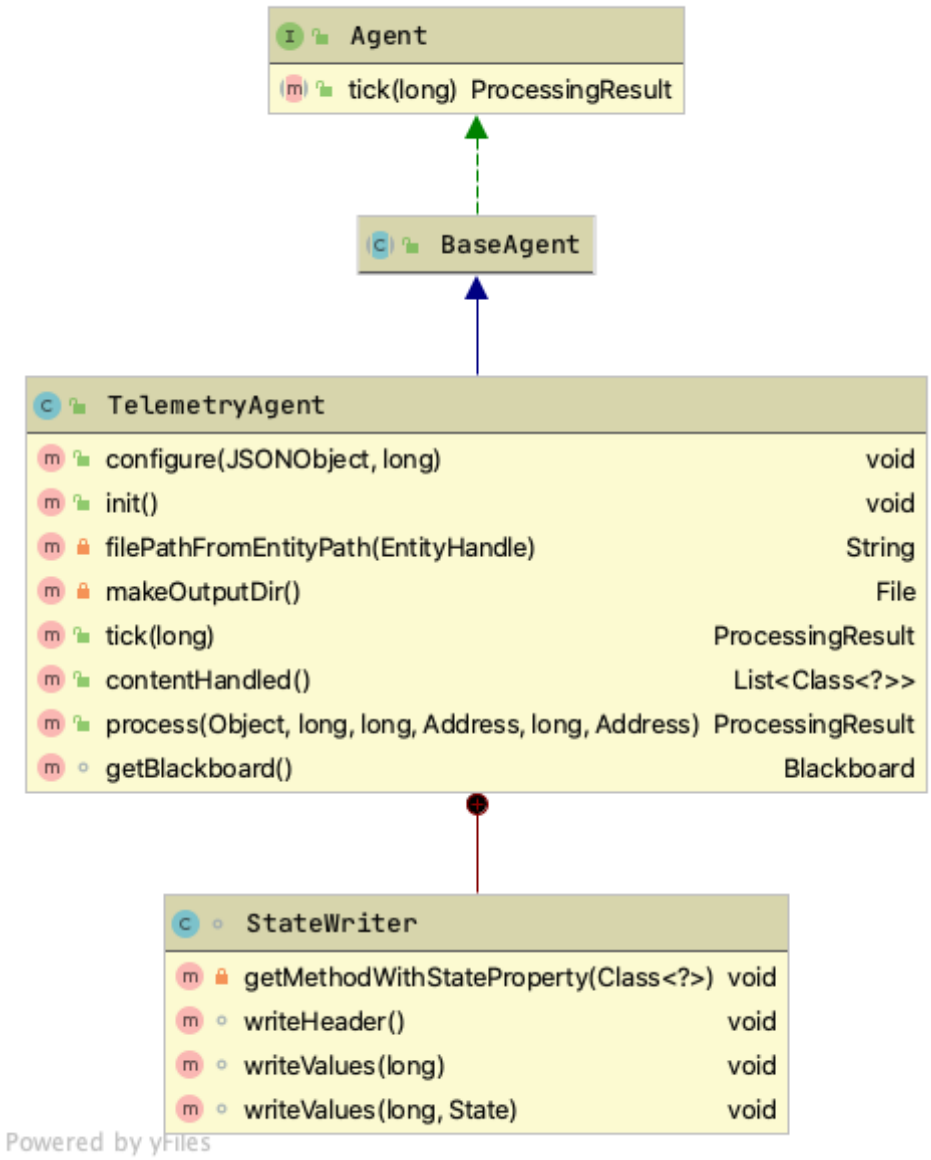

Figure 32. The `TelemetryAgent` class hierarchy

Furthermore, since there is no need to create a separate API, only a single module is required. Code Listing 32 shows the **`build.gradle`** file. Note that the file specifies that the module has an implementation dependency on the **`Blackboard`** API as well as a test dependency on the **`Blackboard`** implementation module.

Code Listing 32. The `build.gradle` File for `TelemetryAgent`

```
ext.moduleName = 'mil.sstaf.telemetry'

dependencies {
    implementation project(':framework:mil.sstaf.core')
    implementation project(
        ':features:support:mil.sstaf.blackboard.api')
    testRuntimeOnly project(
        ':features:support:mil.sstaf.blackboard.inmem')
}
```

The dependencies on the **`Blackboard`** modules arise because **`TelemetryAgent`** declares a requirement for a **`Blackboard`** through a **`Requires`** annotation, as shown in the excerpt in Code Listing 33. **`TelemetryAgent`** uses this reference to access the **`Blackboard`** to read the values to record. The test dependency is required because an implementation of **`Blackboard`** must be supplied to the **`TelemetryAgent`** during the tests.

Code Listing 33. Excerpt from `TelemetryAgent`

```
public class TelemetryAgent extends BaseAgent {
    public static final String FEATURE_NAME = "Telemetry Agent";
    public static final int MAJOR_VERSION = 0;
    public static final int MINOR_VERSION = 1;
    public static final int PATCH_VERSION = 0;

    public static final String CK_STATES_LIST_= "statesToRecord";

    // ...

    @Requires
    Blackboard blackboard;

    public TelemetryAgent() {
        super(FEATURE_NAME, MAJOR_VERSION, MINOR_VERSION,
            PATCH_VERSION, true, "CSV telemetry writer");
    }
    // ...
}
```

4.2.4 Configuration and Initialization

Unlike the **`Blackboard`**, **`TelemetryAgent`** does require configuration and initialization. Configuration is necessary to set the keys that the agent will use to retrieve the desired values. Initialization is required to prepare the agent to write the values.

Code Listing 34 shows an example JSON configuration that sets up a **`TelemetryAgent`**. Recall that the features array specifies the features to be added to an entity. The configurations object defines the configuration parameters that will be provided to the **`Feature`**. The **`FeatureHandler`** provides the **`JSONObject`** associated with the key "Telemetry Agent" to the **`TelemetryAgent`** when it is loaded. Thus, **`TelemetryAgent`** records the values associated with "Aim Internals" and "Muscle Metrics".

Code Listing 34. `TelemetryAgent` Configuration

```
{
  "features": [
    {
      "featureName": "Telemetry Agent",
      "majorVersion": 0,
      "minorVersion": 1,
      "requireExact": "false"
    }
  ],
  "configurations": {
    "Telemetry Agent": {
      "statesToRecord": [
        "Aim Internals",
        "Muscle Metrics"
      ]
    }
  }
```

Code Listing 35 shows the **`configure`** method in **`TelemetryAgent`**. The method uses the **`iterateOverObjs`** method to retrieve the list from the configuration object. It then iterates over the objects in the list and if the object is a **`String`**, it is added to the list of **`Blackboard`** keys to use. The value of **`CK_STATES_LIST`** is "statesToRecord", which corresponds to the key in the configuration.

Code Listing 35. The `configure` Method

```
@Override
public void configure(JSONObject configuration, long randomSeed) {
    super.configure(configuration, randomSeed);
    JSONUtilities.iterateOverObjs(configuration, CK_STATES_LIST,
        object -> {
            if (object instanceof String) {
               String stateName = (String) object;
               stateKeys.add(stateName);
            }
    });
}
```

Code Listing 36 shows the **`init`** method in **`TelemetryAgent`**. This method performs two actions. First, it creates the directory into which the CSV files will be written. Then, for each key in the **`stateKeys`** list it creates a new **`StateWriter`**. Each **`StateWriter`** instance is responsible for the following:

1. Creating the output file.
2. Retrieving the object from the **`Blackboard`** with the key.
3. Finding the fields in the object that are marked with **`StateProperty`** annotations.
4. Writing the column headers.
5. Writing the **`StateProperty`** values to the output file.

Code Listing 36. The `init` Method

```
@Override
public void init() {
    super.init();
    try {
        File outputDir = makeOutputDir();
        stateKeys.forEach(key -> writerMap.put(key,
            new StateWriter(key, outputDir)));
    } catch (FileNotFoundException e) {
        throw new SSTAFException(e);
    }
}
```

The final method to discuss is the **`tick`** method shown in Code Listing 37. This method is simple. On each invocation, it iterates over the **`StateWriter`** instances and has each process its object.

Code Listing 37. The `tick` Method

```
@Override
public ProcessingResult tick(long currentTime_ms) {
    if (logger.isDebugEnabled()) {
        logger.debug("{}:{} - Ticking at {}", ownerHandle.getPath(),
            featureName, currentTime_ms);
    }
    writerMap.values().forEach(writer ->
        writer.writeValues(currentTime_ms));
    return ProcessingResult.empty();
}
```

4.3 `ManeuverEntityAgent` and `ManeuverCentralAgent`

In the final example, I present two features that are designed to work together, the **`ManeuverEntityAgent`** and the **`ManeuverCentralAgent`**. The purpose of the **`ManeuverEntityAgent`** is to maintain the position and velocity of individual entities. The purpose of the **`ManeuverCentralAgent`** is to track all of the entities and update each entity on the position of the others. The **`ManeuverCentralAgent`** resides in the **`EntityController`**,

These agents were developed to demonstrate how to provide state information from one entity to all of the other entities in a way that did not violate the strict order-of-operations rules required to ensure repeatability. These agents were motivated by an early sub-model that accessed state data from other entities directly. As stated previously, direct access makes it difficult to guarantee repeatability under multiple threads.

Since these features are simple demonstration models, the environment in which the entities maneuver is trivially simple. Entities maneuver on an infinite, unobstructed plane.

4.3.1 Requirements

The requirements for the two agents are simple. For the **`MobilityEntityAgent`**, they are the following:

- Accept messages to set the current position, heading and speed of the **`Entity`**.
- Retain the current position, heading and speed of the **`Entity`** on a two-dimensional surface.
- On every tick, calculate the new position of the **`Entity`** based on its current position and velocity.
- On every tick, provide the **`ManeuverCentralAgent`** with the current maneuver state of the **`Entity`**.
- Accept and process messages to provide the current maneuver state of the **`Entity`**.

For the **`ManeuverCentralAgent`**, the requirements are the following:

- Accept messages for the current maneuver state of all entities in the simulation.
- On each tick, provide a collection of all of the maneuver states to every entity.

4.3.2 Build Configuration

`EntityController` process messages and perform actions on every simulation tick. Thus, the appropriate **`Feature`** type is **`Agent`**.

We also see that the two agents accept messages and send them back and forth to each other. This indicates that we should have an API module that defines the message and response classes. It should define any classes shared between the two agents as well. Thus, with the API module and the two agents we have three modules: **`mil.sstaf.maneuver.api`**, **`mil.sstaf.maneuver.entityagent`** and **`mil.sstaf.maneuver.centralagent`**.

Code Listing 38 shows the build configuration for the API module. The build configuration for **`ManeuverEntityAgent`** and **`ManeuverCentralAgent`** are shown in Code Listings 39 and 40, respectively. As should be expected, the API module depends only on the SSTAF core module. The modules for the two agents depend on SSTAF core, the API module and the **`Blackboard`**. It is interesting that the **`ManeuverCentralAgent`** lacks a **`testImplementation`** line. This is because I violated TDD and forgot to write tests for that module!

Code Listing 38. Build Configuration for the Maneuver API

```
ext.moduleName = 'mil.sstaf.maneuver.api'
dependencies {
    implementation project(':framework:mil.sstaf.core')
}
```

Code Listing 39. Build Configuration for `ManeuverEntityAgent`

```
ext.moduleName = 'mil.sstaf.maneuver.entityagent'

dependencies {
    implementation project(':framework:mil.sstaf.core')
    implementation project(
        ':features:military:mil.sstaf.maneuver.api')
    implementation project(
        ':features:support:mil.sstaf.blackboard.api')
    testImplementation project(
        ':features:support:mil.sstaf.blackboard.inmem')
}
```

Code Listing 40. Build Configuration for `ManeuverCentralAgent`

```
ext.moduleName = 'mil.sstaf.maneuver.centralagent'

dependencies {
    implementation project(':framework:mil.sstaf.core')
    implementation project(
        ':features:support:mil.sstaf.blackboard.api')
    implementation project(
        ':features:military:mil.sstaf.maneuver.api')
}
```

4.3.3 The API Module

To support the requirements to accept, process and dispatch messages, the API module includes six classes for use as message content. The first three are the **`Position`**, **`Speed`** and **`Heading`** classes. These classes are shown in Figures 33, 34 and 35, respectively. As expected, these classes hold the position, speed and heading values for the **`Entity`**. They can be used to set the maneuver state of the **`Entity`**.

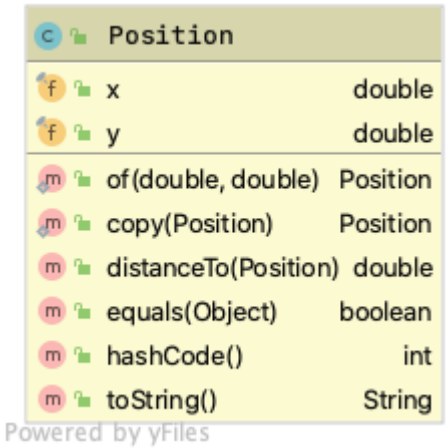

Figure 33. `Position` class

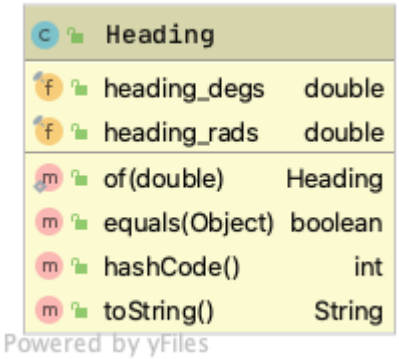

Figure 34. `Speed` class

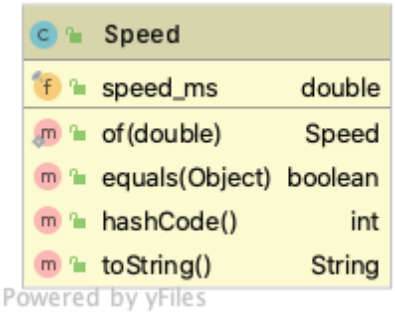

Figure 35. `Heading` class

The fourth class, **`ManeuverState`**, is used to bundle **`Position`**, **`Speed`** and **`Heading`** objects. Like the previous classes, **`ManeuverState`** can be used both to set state and report state. It is shown in Figure 36.

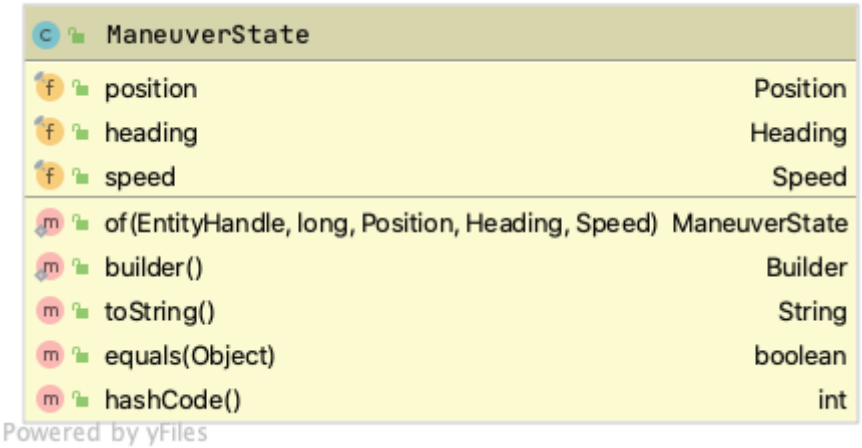

Figure 36. `ManeuverState` class

The fifth class is the **`ManeuverStateMap`** class. This is the class sent from the **`ManeuverCentralAgent`** to every **`Entity`**. It contains a map of **`EntityHandles`** to **`ManeuverState`** objects to provide the required state information. Figure 37 shows the **`ManevuerStateMap`**.

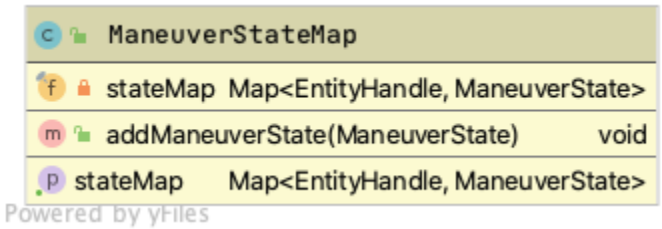

Figure 37. `ManeuverStateMap` class

The sixth class, **`ManeuverStateQuery`** is used to request the current **`ManeuverState`** from an **`Entity`**. It is simply an extension of **`Object`** so that it can be used in an **`instanceof`** test.

The API module also contains an interface that enables other **`Features`** within the **`Entity`** to access the **`ManeuverEntityAgent`** capabilities directly and load them using the **`Requires`** construct. Figure 38 shows the **`ManeuverProvider`** interface.

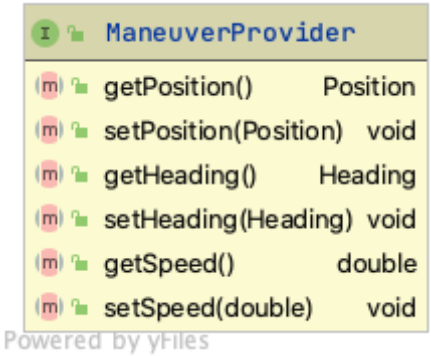

Figure 38. `ManeuverProvider` class

4.3.4 Implementation

Implementation of the two agents is uncomplicated. Neither class supports JSON configuration and neither requires special initialization.

For the **`ManeuverEntityClass`**, we need to specify the messages it supports. Thus, we have the **`contentsHandled`** method shown in Code Listing 41. The **`Heading`**, **`Position`**, **`Speed`** and **`ManeuverState`** objects set the state of the **`Entity`**. The **`ManeuverStateQuery`** requests the current state. The **`ManeuverStateMap`** message provides the **`Entity`** with the **`ManeuverStates`** for all of the other entities.

Code Listing 41. The `ContentHandled` Method

```
@Override
public List<Class<?>> contentHandled() {
    return List.of(Heading.class, Position.class, Speed.class,
        ManeuverStateQuery.class, ManeuverState.class,
        ManeuverStateMap.class);
}
```

As stated previously, these two agents are designed to work together. To show how they do so, I walk through the four methods that perform the collaboration. These are the **`tick`** and **`process`** methods in both the **`ManeuverEntityAgent`** and **`ManeuverCentralAgent`**. These four methods are all executed within one invocation of the **`tick`** method in the **`EntityController`**.

Recall from Section 3.8 the order of execution for **`tick`** and **`process`** for **`Entity`**-level features and **`EntityController`** features. The order is the following:

1. **`Entity`** feature **`tick`**
2. **`EntityController`** feature **`process`**
3. **`EntityController`** feature **`tick`**

4. **`Entity`** feature **`process`**

The **`tick`** method for the **`ManeuverEntityAgent`** is shown in Code Listing 42. The method begins by checking that the position and heading have been initialized. It then updates the position of the **`Entity`** and creates a new **`ManeuverState`**. Next, it looks in the **`Blackboard`** for the **`EntityHandle`** for the **`EntityController`**. The handle is needed to produce the address for the outgoing message. If the handle for the **`EntityController`** is found, an **`EntityEvent`** is created to hold the **`ManeuverState`** object and carry it to the **`EntityController`**. An **`EntityEvent`** is used rather than an **`EntityAction`** because the **`ManeuverState`** is associated with a specific time. A **`ProcessingResult`** that contains the **`ManeuverState`** is returned.

Code Listing 42. The `ManeuverEntityAgent` Tick Method

```
@Override
public ProcessingResult tick(long currentTime_ms) {
    logger.trace("{} ticking at {}", getInfoString(), currentTime_ms);
    double deltaT = currentTime_ms - lastTimeStamp_ms;
    lastTimeStamp_ms = currentTime_ms;

    if (position == null || heading == null) {
        throw new IllegalStateException(
            "Initial Position and/or Heading are not set!");
    }
    position = updatePosition(position, heading, speed, deltaT);
    ManeuverState state = updateManeuverState(currentTime_ms);

    if (centralAgentHandle == null) {
        Optional<EntityHandle> optHandle =
            blackboard.getEntry("SYSTEM:EntityController",
            currentTime_ms, EntityHandle.class);
        optHandle.ifPresentOrElse(
                obj -> centralAgentHandle = obj,
                () -> logger.warn(
                     "EntityController not set"));
    }

    if (centralAgentHandle == null) {
        logger.debug("{} centralAgent handle is null, "
            +   "returning empty ProcessingResult", getInfoString());
        return ProcessingResult.empty();
    } else {
        EntityEvent.Builder builder = EntityEvent.builder();
        builder.withEventTime_ms(currentTime_ms);
        builder.withSource(Address.makeAddress(ownerHandle,
            getName()));
        builder.withDestination(
            Address.makeExternalAddress(centralAgentHandle));
        builder.withContents(state);
        EntityEvent event = builder.build();
        logger.trace("{} returning state {}", getInfoString(), state);
        return ProcessingResult.of(event);
    }
}
```

After all of the entities have completed their tick, all of the messages generated are routed to their destinations. Consequently, all of the **`ManeuverState`** messages will be queued for processing in the **`EntityController`**.

The next method to be invoked is the **process** method in the **ManeuverCentralAgent** that is assigned to the **EntityContoller**. Code Listing 43 shows the source for the method. The method is simple. If the received object is a **ManeuverState**, it is added to a **ManeuverStateMap** that is retained by the agent.

Code Listing 43. The ManeuverCentralAgent Process Method

```
@Override
public ProcessingResult process(Object arg, long scheduledTime_ms,
    long currentTime_ms, Address from, long id, Address respondTo) {
    if (arg instanceof ManeuverState) {
        ManeuverState maneuverState = (ManeuverState) arg;
        if (maneuverStateMap == null) {
            maneuverStateMap = new ManeuverStateMap();
        }
        maneuverStateMap.addManeuverState(maneuverState);
    } else {
        throw new SSTAFException(arg.getClass()
            + " is not supported by this Handler");
    }
    return ProcessingResult.empty();
}
```

The next method to be invoked is the **tick** method for the **ManeuverCentralAgent**. Code Listing 44 shows this method. This method is also simple. It iterates over all of the keys in the **ManeuverStateMap** and uses those **EntityHandle** objects to form addresses for new messages. A message that contains the **ManeuverStateMap** is created for each **Entity** in the map. These messages are returned in the **ProcessingResult** and then routed to the entities.

Code Listing 44. The ManeuverCentralAgent Tick Method

```
@Override
public ProcessingResult tick(long currentTime_ms) {
    List<Message> output = new ArrayList<>();
    if (maneuverStateMap != null) {
        maneuverStateMap.getStateMap().keySet().
        forEach(entityHandle ->
            output.add(prepareMessage(entityHandle)));
    }
    maneuverStateMap = null;
    return ProcessingResult.of(output);
}
```

The final method is the **`ManeuverEntityAgent.process`** method shown in Code Listing 45. It handles the **`ManeuverStateMap`** objects by registering them in the **`Blackboard`**. In addition to the **`ManeuverStateMap`**, the process method handles the classes that change maneuver settings. It also responds to direct requests for maneuver state.

Code Listing 45. The `ManeuverEntityAgent` Process Method

```
@Override
public ProcessingResult process(Object arg, long scheduledTime_ms,
    long currentTime_ms, Address from, long id, Address respondTo) {

    logger.debug("{} processing {} value = '{}'",
            getInfoString(), arg.getClass(), arg);

    boolean update = true;
    if (arg instanceof ManeuverState) {
        ManeuverState maneuverState = (ManeuverState) arg;
        logger.trace("{} updating with {}", getInfoString(),
            maneuverState);
        this.position = maneuverState.position;
        this.heading = maneuverState.heading;
        this.speed = maneuverState.speed;
        logger.trace("{} state = {} {} {}", getInfoString(),
                this.position, this.heading, this.speed);
    } else if (arg instanceof Position) {
        this.position = (Position) arg;
        logger.trace("{} state = {} {} {}", getInfoString(),
                this.position, this.heading, this.speed);
    } else if (arg instanceof Heading) {
        this.heading = (Heading) arg;
        logger.trace("{}, state = {} {} {}", getInfoString(),
                this.position, this.heading, this.speed);
    } else if (arg instanceof Speed) {
        this.speed = (Speed) arg;
        logger.trace("{} state = {} {} {}", getInfoString(),
                this.position, this.heading, this.speed);
    } else if (arg instanceof ManeuverStateQuery) {
        double deltaT = currentTime_ms - lastTimeStamp_ms;
        updatePosition(position, heading, speed, deltaT);
    } else if (arg instanceof ManeuverStateMap) {
        update = false;
        blackboard.addEntry("ManeuverStateMap", arg, currentTime_ms);
    } else {
        throw new SSTAFException("Can't process "
            + arg.getClass().getSimpleName());
    }

    if (update) {
        ManeuverState ms = updateManeuverState(currentTime_ms);
        logger.trace("{} returning current state {}", getInfoString(),
            ms);
        Message response = this.buildNormalResponse(ms, id,
```

```
                respondTo);
            return ProcessingResult.of(response);
        } else {
            return ProcessingResult.empty();
        }
}
```

At this point, the major concepts of SSTAF features should be clear. In the next section, I discuss creating an application that uses SSTAF.

5. ASSEMBLING A SIMPLE APPLICATION

To illustrate how to construct a SSTAF-based application, I walk through a simple application that is included in the SSTAF source repository. The application loads a set of units and Soldiers and a set of initial commands and events. The application submits the commands and events to the specified units or Soldiers. The simulation then enters an event-processing loop. The simulation processes one tick at a time with a user-specified interval for a user-specified duration. When the duration has been achieved, the application exits.

The simulation does not produce any results except those that are captured by the telemetry feature. The purpose of the application is to exercise the SSTAF system from top to bottom.

5.1 Build Configuration

Code Listing 46 shows the build file for the application. It is much more complicated than the previous examples. The top half of the file loads the Gradle application plugin and configures it to build the desired application. The second half specifies the dependencies for the application.

Within the dependency block are three groups of dependencies. In the first block, the two SSTAF framework modules are specified. In the second, additional implementation dependencies are declared on the API modules for several features. These dependencies exist because an application can instantiate feature-specific message objects to interact with loaded features. The third group of dependencies are runtime dependencies for the models and support features that are loaded dynamically by the system.

Code Listing 46. Application Build File

```
plugins {
    id 'application'
}

ext.moduleName = 'mil.sstaf.analysis'

apply plugin: 'application'

application {
    mainClassName = 'mil.sstaf.analysis.Main'
    applicationDefaultJvmArgs = ["log4j.configurationFile=/Users/rbowe
rs/Desktop/Projects/Overmatch/sstaf-master/system/application/mil.ssta
f.analysis/src/main/resources/log4j2.xml"]
}

dependencies {
    implementation project(':framework:mil.sstaf.core')
    implementation project(':framework:mil.sstaf.session')

    implementation project(':features:military:mil.sstaf.aim.api')
    implementation project(':features:military:mil.sstaf.injury.api')
    implementation project(':features:human:mil.sstaf.kinematics.api')
    implementation project(
        ':features:military:mil.sstaf.targeting.api')
    implementation project(':features:human:mil.sstaf.physiology.api')
    implementation project(':features:human:mil.sstaf.equipment.api')
    implementation project(
        ':features:human:mil.sstaf.anthropometry.api')
    implementation project(':features:military:mil.sstaf.soldiertasks.
api')

    runtimeOnly project(':features:military:mil.sstaf.aim.dynamic')
    runtimeOnly project(':features:military:mil.sstaf.aim.standard')
    runtimeOnly project(':features:military:mil.sstaf.injury.orca')
    runtimeOnly project(':features:support:mil.sstaf.blackboard.inmem'
)
    runtimeOnly project(':features:human:mil.sstaf.kinematics.simple')
    runtimeOnly project(':features:human:mil.sstaf.equipment.manager')
    runtimeOnly project(
        ':features:military:mil.sstaf.targeting.acquire')
    runtimeOnly project(':features:human:mil.sstaf.physiology.agent')
    runtimeOnly project(
        ':features:human:mil.sstaf.physiology.models.cardiovascular')
    runtimeOnly project(
        ':features:human:mil.sstaf.physiology.models.cognition')
    runtimeOnly project(
        ':features:human:mil.sstaf.physiology.models.energy')
    runtimeOnly project(
```

```
        ':features:human:mil.sstaf.physiology.models.hydration')
    runtimeOnly project(
        ':features:human:mil.sstaf.physiology.models.musculature')
    runtimeOnly project(
        ':features:human:mil.sstaf.physiology.models.respiration')
    runtimeOnly project(
        ':features:human:mil.sstaf.physiology.models.vision')
    runtimeOnly project(
        ':features:military:mil.sstaf.soldiertasks.entity')
    runtimeOnly project(':features:support:mil.sstaf.telemetry')
}
```

5.2 Implementation Classes

The entire application consists of two public classes: **`Main`** and **`AnalysisRunner`**.

The **`Main`** class is simple. Its only responsibilities are to check the command-line arguments, print a usage message if they arguments are bad, create the **`AnalysisRunner`** and start the **`AnalysisRunner`**.

5.2.1 Building the `AnalysisRunner`

The **`AnalysisRunner`** is created and configured using the factory and builder idiom described previously.

Code Listing 47 shows the **`Factory`** constructor in the **`AnalysisRunner.`** The method begins by creating the **`Session`** object by using its **`factory`** method to parse and load the **`JSONObject`** that defines the units and Soldiers.

Once the **`Session`** is loaded, the **`EntityHandles`** are retrieved and a map of paths to handles is built. This is used later when the commands are loaded. The **`Session`** is then added to the **`AnalysisRunner.Builder`** and the method moves on to read the command file.

Code Listing 47. The `Factory` Method

```
private Factory(JSONObject sessionObj, JSONObject configObj) {
    builder = builder();

    Session session = Session.factory(sessionObj).build();
    SortedSet<EntityHandle> handles = session.getEntities();
    handles.forEach(handle ->
        handleMap.put(handle.getPath(), handle));
    builder.withSession(session);

    JSONUtilities.setOption(configObj, "tickInterval",
        Long.class, builder::withTickInterval);
    JSONUtilities.setOption(configObj, "runDuration",
        Long.class, builder::withRunDuration);
    JSONUtilities.iterateOverJSONObjects(configObj, "commands",
        this::processObject);
}
```

5.2.2 Command File and Parsing

Code Listing 48 shows a sample command file. Like all other files used by SSTAF, it is implemented in JSON. The file consists of three elements. The first, "tickInterval", specifies the simulation time between simulation ticks. The second element, "runDuration", specifies how long the simulation will run in simulated time. The third element, "commands", is a list of JSON object blocks that define commands and events to be added to the system and processed.

Code Listing 49 shows an example command file that is loaded by the **`AnalysisRunner`** factory. First, the tickInterval and runDuration values are loaded, then the list provided by the commands element are processed.

The form of each command object varies according to the object created. Two common elements are the recipient and type elements. Recipient specifies which **`Entity`** is to receive the message. It is specified as a path to the **`Entity`**. The type element specifies the class of the object and is used to select how to process the remaining elements in the object.

Code Listing 48. Command File for the Example Application

```
{
  "tickInterval": 1000,
  "runDuration": 36000000,
  "commands": [
    {
      "recipient": "Tango Squad:SL",
      "type": "ReloadTask",
      "weaponName": "M4"
    },
    {
      "recipient": "Tango Squad:SL",
      "type": "UseItemTask",
      "itemName": "M4",
      "dominantArmFraction": 0.45,
      "nonDominantArmFraction": 0.55
    }
  ]
}
```

Code Listing 49. The `ProcessObject` Method

```
private void processObject(JSONObject jsonObject) {
    logger.debug("Processing {}", jsonObject.toString(2));

    makeContents(jsonObject).ifPresent(content -> {
        if (commandIsEvent(jsonObject)) {
            logger.warn("Event is not supported yet");
        } else {
            var cmdBuilder = SSTAFCommand.builder();
            JSONUtilities.setMandatoryOrThrow(jsonObject,
                "recipient", String.class, path -> {
                if (handleMap.containsKey(path)) {
                    cmdBuilder.withRecipient(handleMap.get(path));
                } else {
                    throw new SSTAFException("The recipient path '"
                    + path
                    + "' does not exist in the session configuration."
                    );
                }
                cmdBuilder.withContent(content);
            });
            SSTAFCommand command = cmdBuilder.build();
            builder.withCommand(command);
            logger.info("Queued: {}", command);
        }
    });
}
```

For each object in the commands list, the **ProcessObject** method shown in Code Listing 49 is invoked. If the object defines a **SSTAFEvent**, it is skipped for now. If the object defines a command, another builder is instantiated to build the command. The content of the command is then constructed using the **makeContents** method shown in Code Listing 50. Then, the content is added to the command and the command is added to the **AnalysisRunner** builder.

Code Listing 50. The makeContents Method

```
Optional<Object> makeContents(JSONObject jsonObject) {

    String commandType = jsonObject.getString("type");

    Object rv = null;
    if ("MarchTask".equalsIgnoreCase(commandType)) {
        logger.warn("MarchTask is not supported yet");
    } else if ("UseItemTask".equalsIgnoreCase(commandType)) {
        rv = UseItemTask.factory(jsonObject).build();
    } else if ("ReloadTask".equalsIgnoreCase(commandType)) {
        rv = ReloadTask.factory(jsonObject).build();
    } else if ("AimErrorTask".equalsIgnoreCase(commandType)) {
        rv = AimErrorTask.factory(jsonObject).build();
    }
    return Optional.ofNullable(rv);
}
```

After all of the commands have been read, the **AnalysisRunner** can be constructed. The next step is for the main method to invoke run on the **AnalysisRunner**.

5.2.3 The Run Method

The event loop for the simulation is found in the **run** method of the **AnalysisRunner** class and, as revealed in Code Listing 51, it is not complicated.

The method begins by submitting all of the commands that were read from the command file into the **Session**. Within the **Session**, the messages are routed to the correct entities. Next, the method enters the event loop itself, looping from 0 ms to the **runDuration** at **tickInterval** intervals. Within the loop, the **tick** method for the **Session** is invoked, which in turn invokes the **tick** method in the **EntityController**. Any messages that are returned from **tick** are logged. When the loop completes the method, the application exits.

Code Listing 51. The `Run` Method

```
public void run() {

    for (SSTAFCommand command : commands) {
        session.submit(command);
    }

    logger.info("Run started: tickInterval={} runDuration={}",
        tickInterval, runDuration);
    for (long time_ms = 0; time_ms <= runDuration;
        time_ms += tickInterval) {
        if (logger.isInfoEnabled()) {
            logger.info("Tick: {} -> {}",
                time_ms, Formatter.millisToDHMS(time_ms));
        }
        TickResult tickResult = session.tick(time_ms);
        for (SSTAFResult result : tickResult.getMessagesToClient()) {
            logger.info("{}", result);
        }
    }
    logger.info("Run completed");
}
```

The easiest way to extend the template provided by this application is to feed additional commands and events in through the **`Session`** and do something with the results that come back. I used this approach to develop an application that can use SSTAF to generate static data files for use in legacy models. In that application, each iteration of the event loop performs a status query on each entity. The application recorded the results of those queries as time-dependent capability records in standard file format files.

6. IMPLEMENTED MODELS

The real value of SSTAF is not in the framework, but in the models that it can access and tie together. In this section, I present a summary of the models that have been implemented and can be used within SSTAF. These models have come from numerous organizations including the DEVCOM Soldier Center, the U.S. Army Research Institute of Environmental Medicine and DAC itself.

Figure 39 shows the models expressed in terms of their maturity and the degree to which they have been integrated into the SSTAF environment. For discussion, the models are divided into three categories: (1) accredited models, (2) models under development and (3) experimental models. The three categories are indicated by the three colors of stars.

The accredited models and models under development have their origins outside of SSTAF. The SSTAF project is working to make those models accessible within the environment and work with other models. The experimental models are models that I developed as proofs of concept and to help test the system. These models provide behavior that appears realistic, at least on the surface, but they are not necessarily valid. As we go forward, these models could be corrected, extended or even discarded.

Because this document is focused on the SSTAF architecture rather than the models, the descriptions are brief. Future documents will descript the details of each model, including verification and validation support.

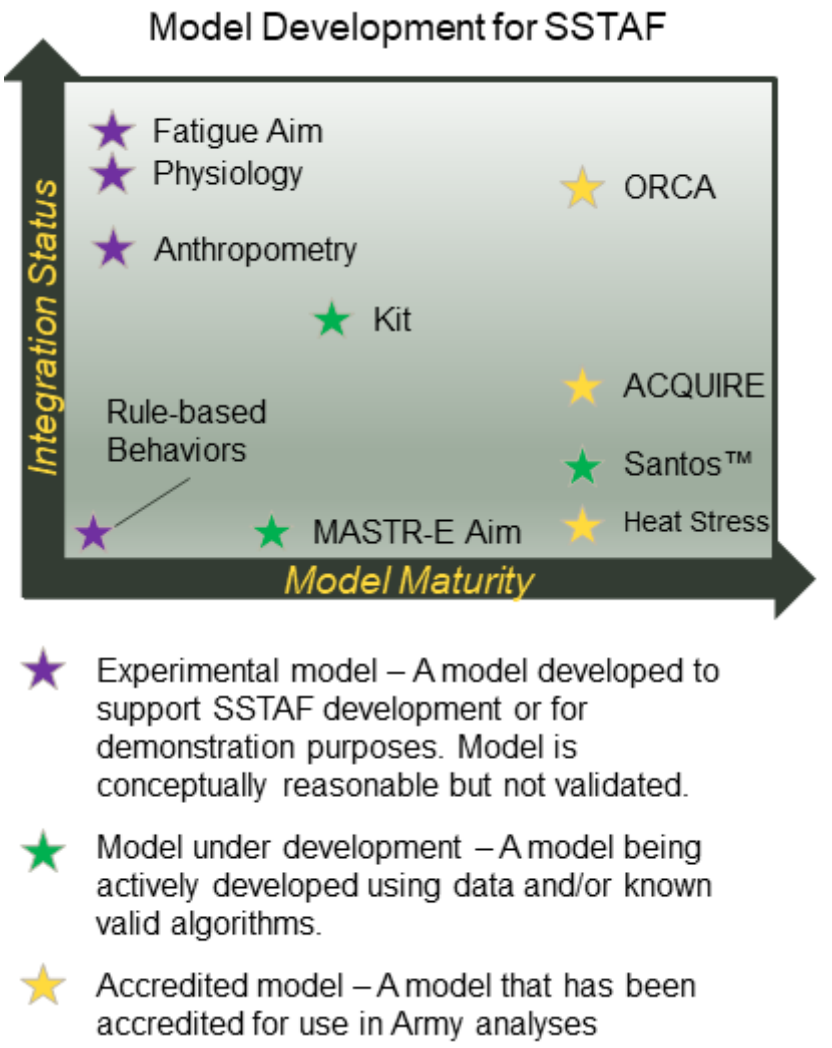

Figure 39. SSTAF models

6.1 Operational Requirements-based Casualty Assessment

The first model integrated into SSTAF was DAC's Operational Requirements-based Casualty Assessment (ORCA) model. ORCA predicts injury and incapacitation for several battlefield insult types at varying degrees of fidelity. These include penetration, blast, thermal, directed energy and toxic gas threats.

The current version is implemented as a simple **`Feature`** and thus cannot respond to messages. Therefore, in order for it to be used within an analysis, it must be loaded into a **`Handler`** or **`Agent`** that can interact with it through the **`InjuryProvider`** interface.

Currently, SSTAF can create insults from ballistic threats and access the injury and incapacitation state of the Soldier. Injury due to blast will be added soon.

6.2 Anthropometry

A simple, experimental **`Feature`** was developed to hold the anthropometric information for the humans and Soldiers in the simulation. The **`Anthropometry Feature`** is a simple collection of the physical characteristics of the person.

The current version includes only a small number of values, specifically the following:

- Sex
- Age
- Height
- Weight
- Arm span
- Handedness

This collection is a small subset of the values available through real data sets such as Anthropometric Survey of U.S. Army Personnel (ANSUR) II. However, these few values are enough to demonstrate the ability of the system to use anthropometric data to affect results.

6.3 Physiology

The **`PhysiologyAgent`** is another experimental model. This model calculates and provides all of the instantaneous physical states of the human. The **`PhysiologyAgent`** is a composite of several other models that handle specific aspects of physiology. These aspects are the following:

- Musculature
- Vision
- Cardiovascular
- Respiration
- Cognition
- Energy
- Hydration

Currently, only the musculature model has any capability, and that capability is limited to calculating arm strength and endurance. The arm strength and endurance model is simple and makes numerous assumptions.

The initial arm strength values are based on the assumption that Soldiers can do pull-ups; therefore, the initial total arm strength value is primarily a function of weight. The dominant arm is able to maintain 85% of total arm strength, while the non-dominant arm can maintain 72% of body weight. Endurance and strength are gradually lost due to carrying a load. Arm endurance

is expressed as the total impulse that the arm can exert before it fatigues. Initial arm for all Soldiers was assumed to be the maximum arm strength maintained for 100 s.

6.4 Equipment Management

Since the primary purpose of SSTAF is to enable trade studies of Soldier equipment, we have implemented a model for specifying the equipment. Each Soldier can be configured with their own equipment, and that equipment is used by the other models to affect performance.

Currently, the **`EquipmentManagment Feature`** supports three types of equipment: packs, guns and magazines. It includes a mechanism for changing the weight of a gun as rounds are expended. The kit model also tracks the number of rounds in a loaded magazine and can prevent a shot from being fired if the gun is empty.

6.5 Fatigue Aim

I used evaluating the effects of fatigue on Soldier aim error as the driver for implementing and demonstrating SSTAF capability. Soldier aim error is the component of shot accuracy that expresses the ability of the Soldier to point the weapon at the right spot to get a hit. It is distinct from other components such as the mechanical accuracy of the weapon or environmental effects.

The issue that we hope to address is that fatigue is underrepresented in combat simulations. In particular, fatigue does not affect the ability of the Soldier to put bullets on adversaries. Additionally, usually there is variability in aim accuracy between Soldiers or between weapons. It is likely, although hard to verify, that modeled accuracy overpredicts real-world performance.

Ideally, we would like to have an aim model that is sensitive to fatigue, that can change during a simulation and that is configurable for each Soldier and each weapon used by that Soldier.

To investigate whether we could improve the aim behavior and test drive SSTAF, I developed an experimental fatigue aim model. This model is a simple physics-based model. It assumes that as the Soldier fatigues, their arms will weaken and the aim error will expand. Also the center of error might drift off of the intended aim point. Figure 40 illustrates the basic concept of the model.

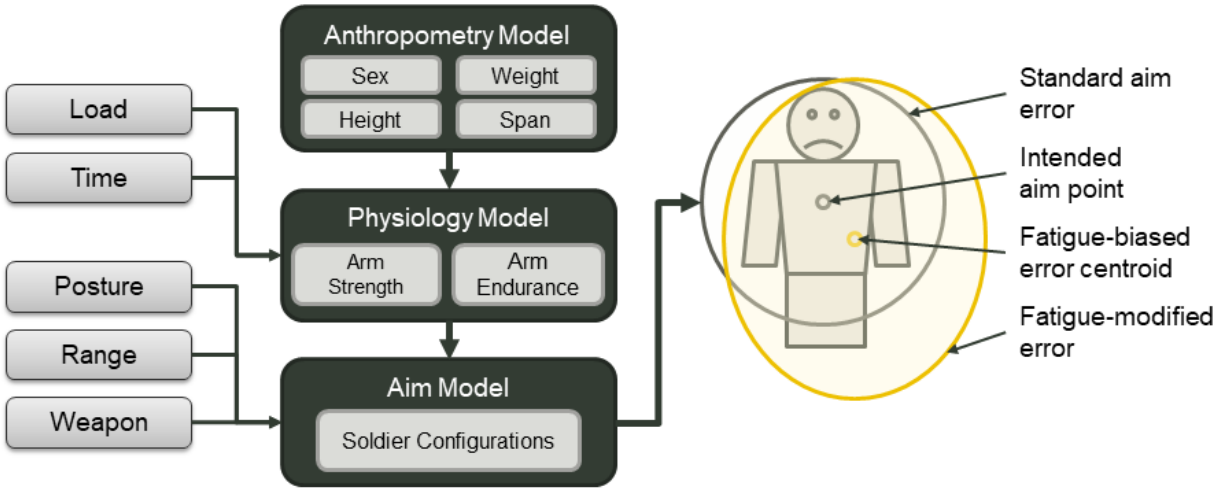

Figure 40. Fatigue aim model

To implement this model, I tied together three experimental models, specifically the anthropometry, physiology and aim models.

Anthropometry specified the physical characteristics of the Soldier. The physiology model used these values to determine an estimate of arm strength and arm endurance. When the simulated Soldier fires a shot, the aim model calculates the aim error. It is calculated based on the arm strength, the weight of the gun, range to the target and Soldier-specific performance parameters.

6.6 ACQUIRE

The ACQUIRE target acquisition and tracking model was implemented early in SSTAF development. ACQUIRE was implemented directly from the Physical Model Knowledge Acquisition Document (PKAD) and the implementation passes the reference tests in the PKAD.

The initial ACQUIRE implementation highlighted several architectural and idiomatic issues with early SSTAF. In particular, it revealed that direct communication between **`Entity`** instances broke repeatability. This resulted in the introduction of the **`EntityHandle`** and elimination of easy access to other **`Entity`** instances.

6.7 Telemetry

As discussed earlier, the **`TelemetryAgent`** provides the ability to retrieve and record state records from objects stored in the **`Blackboard`**.

7. FUTURE WORK

Although the core SSTAF system is stable and usable, there is considerable work still to be done. Most of the work will be on the models and other features that run within SSTAF. Other efforts will focus on enabling the use of SSTAF for analyses.

7.1 Santos

The most significant model that we are working to integrate with SSTAF is the Iowa Technology Institute's Santos™ human kinematics simulation.

Santos is a high-resolution simulation of dynamics of the human muscle and skeletal system. It can predict motion through a task using an optimization strategy that can be constrained by burdens and encumbrances. Santos will provide us with the ability to predict the motion of Soldiers through tasks and maneuvers. It will give us the ability to look at the effects of loads and encumbrances on Soldiers, helping us to identify and quantify some of the negative effects of Soldier equipment.

Our approach with Santos will be to use it as an interactive service. After selecting the avatar based on anthropometry and configuring it based on the kit configuration, we will issue simulation commands to Santos based on the movements required by the combat simulation. After Santos models the motion, we will get several metrics including time to completion, energy expenditure and muscle fatigue. We can then propagate the Santos results to other models.

In order to be able to model the motions a Soldier might be tasked to perform, we are working on developing a library of motion models, to include individual movement techniques. A previous project enabled Santos to be enhanced with models of the motions used when a Soldier executes the exercises required in the Load Effects Assessment Program (LEAP) obstacle course.

7.2 SSTAF-as-a-Service

Although SSTAF is designed to be embedded as a library, at a higher level, another approach to using SSTAF is possible. This approach is to build SSTAF into a web application and make it available as an interactive network service.

Deploying SSTAF as a web service has several benefits. It can make SSTAF available to applications into which direct integration would be difficult. It facilitates integrating SSTAF into cloud-based simulations. It also makes it easier to scale SSTAF analyses by spreading them across multiple computers.

7.3 Squad Operational Value Evaluation using Realistic Metrics and Tactical Capability Hierarchies

Finally, we will move to the top of the stack and look at possible application of SSTAF.

Referring back to the modeling simulation vision that I presented in Section 1.1, Figure 41 shows our vision of a system to meet the requirement for a turn-key analytic system for performing trades space and acquisition analysis of Soldier equipment. We call this system Squad OVERMATCH.

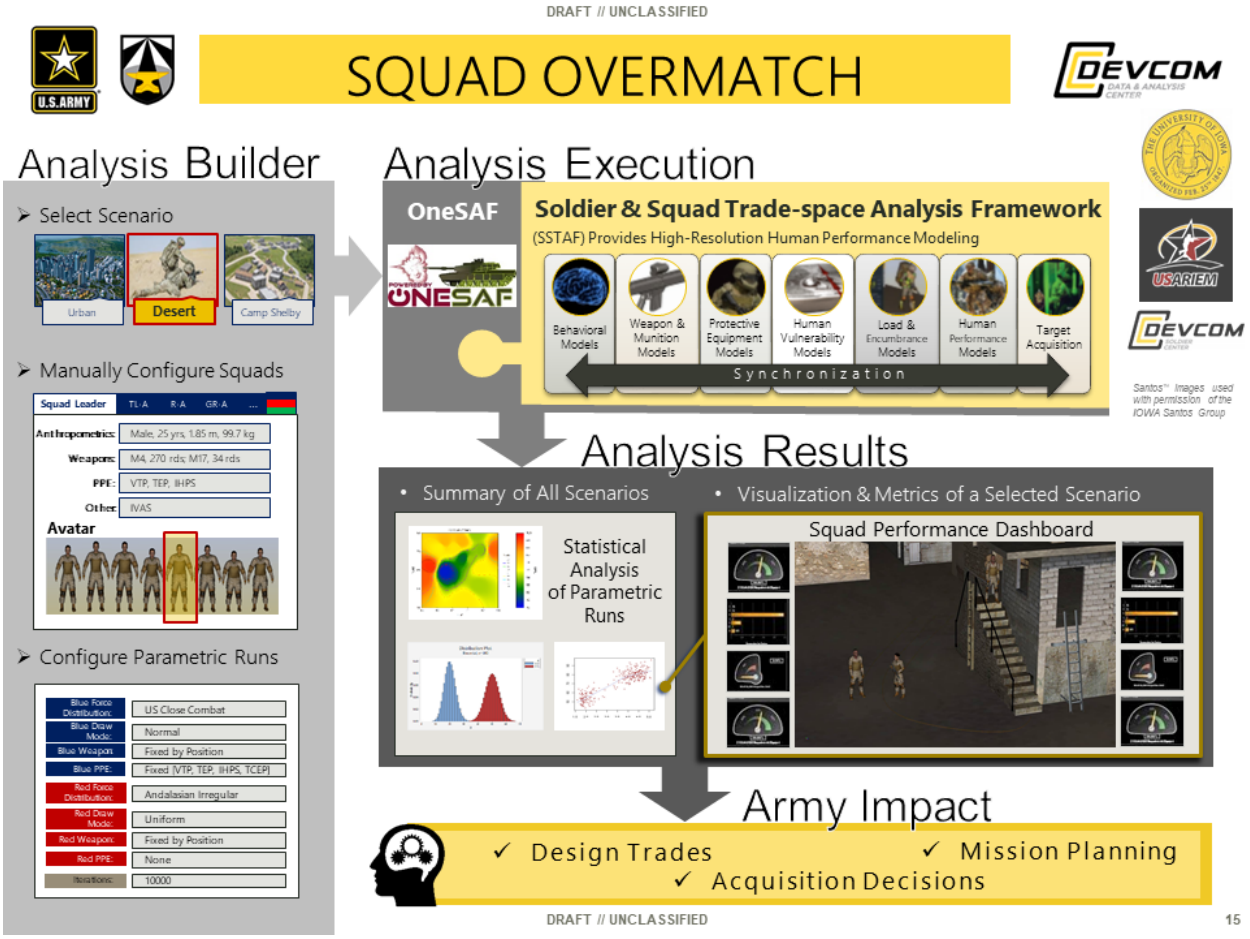

Figure 41. The Squad OVERMATCH system

The Squad OVERMATCH system is centered on SSTAF embedded in OneSAF. This combination provides us with the means to model Soldiers and units in an operational context. We then propose to wrap that system with custom input and output tooling. On the input side, we will have the ability to select scenarios and configure each of the Soldiers in the simulation.

On the output side, we will have statistical tools to dig into the results. We will also leverage Santos visualization components so as to be able to play back Soldiers' activities during the simulation.

To date, we have done some initial planning and research for this project. In particular, the Iowa Technology Institute has been evaluating leveraging some of their existing components for input preparation. More significantly, they have been investigating migrating their visualization system from their current system to using the Unreal Engine. This would facilitate integrating the Santos avatars into other environments.

8. CONCLUSION

SSTAF provides a flexible architecture for developing and integrating human performance and integrating those models into applications. However, despite its capabilities, the real value of SSTAF might be in rallying model developers to use it as a government-owned platform for interactive interoperability. If SSTAF became a widely used platform, progress in Soldier modeling, including digital twinning, would likely accelerate.

One way to maximize adoption of SSTAF is to simplify getting it. Currently, SSTAF is hosted on a government system that is difficult for non-DOD developers to access. This makes it difficult to expand the development of SSTAF models to university and corporate partners. I therefore recommend that the core modules of the SSTAF be reviewed and approved for public release, distributed under an open-source license and hosted on a public development site.

9. REFERENCES AND DOCUMENTS

Bloch, J. (2008). *Effective Java* (2nd ed.). Pearson Education.

Bloch, J., Bowbeer, J., Lea, D., Holmes, D., Peierls, T., & Goetz, B. (2006). *Java concurrency in practice*. Pearson Education.

Helm, R., Vlissides, J., Gamma, E., & Johnson, R. (1994). *Design patterns: elements of reusable object-oriented software.* Pearson Education.

Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Prentice Hall.

Pressman, R. S. (2001). *Software engineering: a practitioner's approach* (5th ed.). McGraw Hill.

Schmidt, D., Stal, M., Rohnert, H., & Buschman, F. (2000). *Pattern-oriented software architectures: patterns for concurrent and networked objects*. John Wiley & Sons.

Appendix A – Building the Soldier and Squad Trade Space Analysis Framework (SSTAF)

1) Squad Operational Value Evaluation using Realistic Metrics and Tactical Capability Hierarchies (Squad OVERMATCH). Obtaining the Code:

   a) The SSTAF source code is currently hosted on the DI2E site. To access to the code:

      i) Obtain a DI2E account.

      ii) Request access to the SSTAF-ReadWrite and SSTAF-STASH-ReadWrite groups. The DI2E system will notify one of the project administrators and they will grant access.

2) Required Tools. The following tools are required to build SSTAF:

   a) Java JDK, Version 11 or later.

   b) Gradle, Version 5.x.

   c) I recommend the use of an integrated development environment. My strong preference is JetBrains IntelliJ/IDEA.

3) Dependencies

   SSTAF carries all dependencies with the project in the repository directory. This was required because DI2E does not allow artifact resolution to access external sites and thus the system could not build using Jenkins continuous integration system on DI2E. There was no apparent mechanism to deploy artifacts to the DI2E artifact repository, so the most expeditious approach was to keep the artifacts in the project.

4) Building the System

   From the SSTAF root directory, invoke "**`gradle build`**".

Appendix B – List of Acronyms

ANSUR	Anthropometric Survey of U.S. Army Personnel
API	application programming interface
CFT	Cross Functional Team
CPU	central processing unit
CSV	comma-separated value
DAC	Data & Analysis Center
DEVCOM	Combat Capabilities Development Command
DOD	Department of Defense
IDE	integrated development environment
JMS	Java Module System
JSON	JavaScript Object Notation
LEAP	Load Effects Assessment Program
MADE	Modernization Application and Data Environment
no-arg	no-argument
OneSAF	One Semi-Automated Force
ORCA	Operational Requirements-based Casualty Assessment
PKAD	Physical Model Knowledge Acquisition Document
SL	Soldier Lethality
Squad OVERMATCH	Squad Operational Value Evaluation using Realistic Metrics and Tactical Capability Hierarchies
SSTAF	Soldier and Squad Trade Space Analysis Framework
TDD	test-driven development
UML	Unified Modeling Language

XML	Extensible Markup Language

Appendix C – Distribution List

ORGANIZATION

DEVCOM Data & Analysis Center
FCDD-DAW-W/R. Bowers
FCDD-DAD-TP/G. Dietrich
FCDD-DAW-W/L. Hall
FCDD-DAW-W/T. Fargus
FCDD-DAW-W/K. Jubb
FCDD-DAW-W/A. Kulaga
FCDD-DAW-W/T. Myers
FCDD-DAW-G/K. Burley
FCDD-DAW-W/J. Collins
FCDD-DAW-W/Z. Steelman
FCDD-DAG-S/J. Acheson
FCDD-DAG-S/R. Vanamburg
FCDD-DAG-S/J. Way
FCDD-DAH-W/A. Boynton
FCDD-DAH-W/J. Sperlein
6896 Mauchly St.
Aberdeen Proving Ground, MD 21005-5071

DEVCOM Army Research Laboratory
FCDD-RLW-B/P. Gillich
FCDD-RLW-B/C. Hoppel
328 Hopkins Rd
Aberdeen Proving Ground, MD 21005-5066

DEVCOM Army Research Laboratory
FCDD-RLD-DCI/Tech Library
2800 Powder Mill Rd.
Adelphi, MD 20783-1138

Defense Technical Information Center
ATTN: DTIC-O
8725 John J. Kingman Rd.
Fort Belvoir, VA 22060-6218

PEO Soldier
PEO-SOLDIER/T. Coleman
5901 Putnam Rd
Ft. Belvoir, VA 22060-5422

DEVCOM Soldier Center
FCDD-SCD-EY/G. Matook
10 General Greene Ave
Natick, MA 01760-2612

DEVCOM Soldier Center
FCDD-SCD-ETA/C. McGroarty
12423 Research Parkway
Orlando, FL 32826