InTEL Software Documentation

Introduction

InTEL - the Interactive Toolkit for Engineering Learning - is an NSF-funded project developed by Georgia Tech to improve the teaching of engineering courses (NSF award #0647915, 2007-2011). The software component of the project allows students to work through many types of Statics problems, aiming to reflect the concepts taught in a basic statics course.

This documentation aims to instruct you (educators and developers) in how to use and extend

the software to better meet your needs.

Solving statics problems

The InTEL software allows students to work through statics problems in a way that reinforces the appropriate type of problem solving. There is a tutorial on the instructions page of the toolkit part of the website that illustrates how to solve for the reactions applied to a body using a free body diagram. This section of the documentation goes through that same exercise and explains some of the terms that are used in the software.

When the student first loads a problem, they see the description mode, which provides pictures of the problem as well as a written problem statement and narrative that contextualizes the problem. Pressing the button “Model and Solve” advances to the next step of the exercise.

For this problem, the first step is the “Select” mode. The available diagrams and modes are visible through the tabbed layout on at the top of the window, so the student can go back to the description at any point. Some exercises (such as 3D or trusses) do not immediately start with the select mode, but have a step in between that the student must complete before they can build free body diagrams.

The pedagogical concept behind the select mode is isolation. The student views the schematic for the entire problem, and selects and isolates a piece of it in order to build a free body diagram for that isolated component. In this problem, the student may select either the forearm, the upper arm, or build a FBD for the entire thing. It is not possible to build FBDs for two force members, such as the bicep muscle in this problem.

The select mode gives a clear view of all pieces of the software’s interface. At the upper left is a window for the tasks, which shows each of the tasks that the user must solve in order to complete the exercise. The tasks are checked off as the student progresses through the problem. Below this is the list of knowns, which are necessary in building FBDs. As values are solved (such as reaction forces and unknown given forces), the knowns list grows to include the new values. On the upper right is the navigation window, which gives buttons to orient the view. At the bottom of the screen is the ModePanel, so named because it changes depending on the current mode.

In the free body diagram mode, the background will disappear and the bodies adjacent to the isolated objects will be visible but grayed out. This is slightly difficult to see in the Arm and Purse problem, but the distinction is apparent in problems such as the Bicycle.

The idea behind the free body diagram mode is to attach all the forces applied to this isolated body from the rest of the world. The student can do this by using the buttons on the mode panel to create forces and moments. When the student wants to proceed, they can click the “Check” button, which performs a check to discern whether the FBD is correctly built. It is only possible

to advance if the diagram is correct.

If there are any problems, the software displays the feedback box to give the student a clue as to what might be wrong. The feedback is calibrated to be useful to the student without giving away the answer in order to encourage learning.

When the FBD is completed, the software advances to the equation mode. This works as a continuation of the FBD mode, so it does not open a new tab at the top of the screen. The mode panel now contains three equations that the student can fill out, representing the equations for static equilibrium.

Ideally, what happens in this stage is that there will be three unknowns that can be solved using the three equations. In the case that there are more than three unknowns then the system will not be solvable, and the student must solve a different diagram before continuing on to this one. In cases where there are less than three unknowns, one of the equations must be redundant, and only two need to be solved for the system to be solved.

When the equations are correctly filled out and the system is solvable, the software numerically solves the equations to find out what the values of the unknowns are. At this point, the solved values are added to the knowns window, and the connectors are marked as solved. This means that when the student works on adjacent diagram, they must use the numerical values instead of the old symbolic terms.

Web Toolkit

The toolkit is the public front end of the software, and has facilities for students, graders, and administrators to perform most tasks through a single web interface.

How students use the toolkit

My Assignments

This gives the list of all the problems which are currently assigned. If the student is logged in, this will only display problems which are assigned and are within their start and end dates. If the student does not see anything here, it is likely that no problems are assigned to their section.

If no one is logged in, this page will display all available problems so that anyone can browse and explore the problems we offer. A message will appear at the top of the page warning on the that no grade is being recorded due to not being logged in.

Instructions

The instructions page provides some basic information for students on how to use the applet and offers answers to some frequently asked questions.

Give us feedback

This link opens a window using javascript that allows a student to give essentially spontaneous feedback. This feedback is logged into the database and is visible by site administrators.

Email site support

This link directs to an email specified in the PHP file model/baseinclude.php
Institutions wishing to deploy InTEL will probably want to direct this to someone who will be responsible for technical support.

Types of problems

The toolkit supports the ability to display both problems that use the integrated statics software, as well as both Flash and Java problems that are standalone. Examples of the latter group are “See-Saw Manipulatable,” “Merry-Go-Round Manipulatable,” and “Learning Connectors”. These are generally referred to as the manipulatable problems.

The important distinction between these is that we do not have support for monitoring whether or not students have completed manipulatable problems. These can only be effectively offered as practice problems.

Toolkit for graders and instructors

Graders and instructors have access to several additional panels that handle assignments and submissions.
The current implementation allows only the owner of a class to view submissions and manage assignments for that class. Frequently faculty will create a single account for themselves and their graders.

Site administrators are allowed to access all submissions and edit all assignments. This role is useful for technical support, but may not be desirable for instructors who may not want to have to see information from others’ classes.

View Submissions

This panel allows graders to view and search through submissions by both class and problem. Each row in the view displays the following information:

- name
- email address
- assignment
- assignment type (assignment, extra credit, or practice)

- class (to which the student belongs)
- status (how far along the student is in this problem)
  If the student has started, there is also a link to view their progress.
- date and time of the submission

When users work on assignments, their progress is automatically saved. Viewing a student's progress pulls up a view of the assignment and shows the problem exactly as the student would see it, based on the work that has been saved up until that point. Viewing a student's progress will not actually affect their saved data, though. This can be useful when a student has become stuck on a problem and is either asking for advice or suspects that there may be a bug in the software.

Manage Assignments

This page allows an instructor to create assignments and edit existing ones.

To create an assignment, you must select the class, problem, type of assignment, and the begin and end times for when the assignment is open. The fields for beginning and end dates can accept both dates and times. With no time specified, the date will default to 12 a.m. A valid input for an open or close date can look something like this "8:00 am 07/15/11" and can easily be edited using this interface.

It is easy to edit assignments as well. If any part of the assignment is entered incorrectly or needs to be modified for another reason, it can be changed by pressing the "edit" link on any of the rows for the assignments. This allows you to change all parts of the assignment, the class, problem, type, and open and close dates.

It is important to be aware that students will not be able to see problems unless the time is between the open and close dates.

An assignment must also be given to a valid class, so it is not possible for an instructor to create an assignment until a class is created.

Manage Classes

A new instructor will not have any classes available until a new class is added. These are typically used to represent sections, especially if the sections have different assignments or different due dates. An instructor can only add a class for themselves, but admins can create classes and give them to any instructor.

Manage Accounts

Administrators may also manage accounts, and appoint a new user to be an instructor or an admin without needing to go into the database.

Nuts and bolts

This subsection describes the low-level elements of how the web toolkit works and is primarily going to be useful for administrators.

How progress is saved

When a student is logged in and works on a problem, their progress is sent back to the server and saved. The way this works is that after the student makes an action, the Java software creates a serial representation of the user's state, then uses HTTP POST to submit it to toolkit/auto_postAssignment.php.

Each of these transactions is very quick and the data posted is fairly small (generally less than 10k apiece). This data goes into the app_user_assignment MySQL table.

When a student refreshes their browser window, or stops and opens the assignment again, their old state will be posted as a parameter to the exercise.

Logger data

The Java software will also send its logs to the server to help diagnose any bugs that students may run into. Much like the posting assignment progress, this happens automatically in the background. Log data is submitted to toolkit/auto_postLogger.php, and is stored in the app_problem_usage_log table in the MySQL database. A lot of logger data will be generated over the course of students working on problems, so it is occasionally useful to flush out the table if it gets too big.

Logger data is visible to administrators on the toolkit. The logger data page lists individual sessions that users have had with the software, showing the user's name, the problem, and the date and duration of their work on this particular session.

Looking inside one of these sessions shows the individual log messages, which gives the message's level, the class and method that generated the message, and the message's content. Most log messages are "INFO", however "WARN" and "SEVERE" messages may also occur if there are problems.

One of the first log messages displays the user's system properties. This does not contain any sensitive or personal information, but does show the user's browser and operating system, as well as their version of Java, which may be useful for diagnosing problems.

Logger data may also be analyzed to examine how students solved certain problems, which may be of pedagogical interest to some parties.

Folder structure

The toolkit folder has several important parts. The structure is based off of a Model/View/Controller infrastructure. Most of the pages at the base level of the toolkit simply include the contents of a php file in the view folder.

Subfolders:

- **admin:** This contains initvars.php, which initializes the variables to make the database connection. In the released source code, this information is empty. Developers will need to create their own database login to make sure the software works on their site.
- **controller:** This folder contains the action pages that are used for deleting classes and assignments.
- **images:** This contains the preview images that are used in the view problems page.
- **js:** This contains javascript sources that are used in the toolkit. The two sources that are used are jquery and sortable, the latter of which is used for making sortable html tables.
- **model:** This folder contains library files that are used for expedience sake to interact with the database. The file “baseinclude.php” sets the email address for technical support, and also contains the URL for the toolkit, as well as the toolkit’s file path on the filesystem. This information needs to be updated when deploying on a new server, otherwise the pages will redirect to the Georgia Tech website.
- **resources:** Contains some miscellaneous images.
- **styles:** Contains the CSS stylesheet for the toolkit pages.
- **view:** This contains the bulk of information for how the toolkit appears to users. The header and footer files establish the navigation and perform boilerplate user account operations.

MySQL administration

The MySQL database structure and some starting data is included in the toolkit folder.

It is important to note that MySQL is the only way to make a user an administrator if no other administrators exist in the system. The way to do this is to set that user’s user_type_id to 4 in the app_user table.

It is useful to point out here that the toolkit commands to delete assignments and classes do not actually remove them from the database, but they instead set the flag is_active to 0.

Tables:

- **app_assignment:** Records all assignments that are given to classes.
- **app_assignment_type:** A naming table that gives names to the assignment types.
- **app_class:** Records all classes, and has a reference to the class owner.
- **app_feedback:** Stores all of the feedback that is recorded in the “give feedback” prompt.
- **app_problem:** Records all types of problems. This contains several important pieces of

information.
Each problem has a name, description, and an instructor-only description. Problems must also specify the jar file name for the problem (located in the applet folder), as well as the class name for the Exercise that makes up this problem. Problems must also specify their problem type, which may be either “java”, “simplejava”, or “flash”. Problems should specify their width and height, though all exercises built through the statics module should be run at 1100x768.
The field ordseq denotes the general order in which problems should be displayed on the view assignments page. The order is generally given according to the subject and how advanced the problem is supposed to be.

- **app_problem_arg:** (unused)
- **app_problem_tag:** (unused)
- **app_problem_type:** Naming table for problem types (assignment, extra credit, and practice).
- **app_problem_usage_log:** Contains all the logger data. This table can get to be massive and may need to be flushed occasionally.
- **app_problem_usage_sessions:** Records the sessions for logger data.
- **app_submission_status:** Naming table for submission status flags counters.
- **app_user:** Records all users. Each user has a uuid (universal unique id), email address, encrypted password information, name, and account type.
- **app_user_assignment:** Contains information about user completion of assignments. This is the information that appears in the View Submissions page.
- **app_user_class:** Relational table connecting users to classes. This relationship is many-to-one, but is placed in a separate table for ease of management.
- **app_user_type:** Naming table for the different types of users (anonymous, student, instructor, admin).
- **survey_credit:** Used to record survey completion specific to data collection for the InTEL project.

Getting started with the code

This section assumes a base level of programming knowledge and familiarity using IDEs.

Getting ready to work

This section describes the steps that are necessary for running the software in a development environment.

Checking out the code (Subversion)

This section assumes that you are a member of an institution which has a Subversion repository configured to hold the project source code, and that you have the address for it. Checking out the code and building the project is the first step toward being able to work with it and create new functionality or exercises.

To check out the code, you need a Subversion client, such as TortoiseSVN (available at http://tortoisesvn.tigris.org/). TortoiseSVN is the client we have used the most frequently in developing the software, but it is Windows only. There are Mac SVN clients (such as svnX, http://www.lachoseinteractive.net/en/community/subversion/svnx/features/), but detailed instructions for using them is outside the scope of this document.

<sshot of checkout window in TortoiseSVN>
To check out the source code, create an empty folder for the project. Inside of that folder right-click and select “SVN Checkout...”. Using TortoiseSVN, enter in your source URL as illustrated

in the screenshot. If everything is configured correctly, you should see the folder populated with the the files from version control.

Checking out the code (Github)

Another way to get the source code is to check it out using Git instead of Subversion. The code is hosted on Github, for this purpose.

How to get started with Github

1. Create an account at http://github.com. The email address you enter here is the address Git (and Github) will use to identify you.
2. Install the Git client software for your operating system (http://git-scm.org).
    a. Windows - http://help.github.com/win-set-up-git/)
    b. OS X - http://help.github.com/mac-set-up-git/ and
    c. Linux - http://help.github.com/linux-set-up-git/.
3. These guides also have instructions for creating your SSH keypairs, but here is a quick overview explanation:
    a. Generate a pair of keys; one is public and the other is private. (Github instructions provide more detail about how to do this.)
    b. Copy the public key to your Github profile. The private key is kept private.
    c. When interacting with Github repositories (clone, pull, push, branch etc), Github uses the private key stored on your computer to validate your identity, in combination with the public key you have provided.

Here is an overview of some basic Git operations:

- **git clone** - Clone an existing repository, hosted somewhere, onto your local machine. In practical terms, this means that Git will create a new directory with the same name as the repository in your current directory. It will then download the repository into that new directory. This is analogous to svn checkout.
- **git pull** - Downloads changes to the remote hosted repository and merges them with your local copy. This is analogous to svn update.
- **git add** - Stages changes to local repository for commit.
- **git commit** - Commits changes to your LOCAL repository.
- **git push** - Pushes changes to the remote hosted repository. Analogous to svn commit.
- **git branch** - Creates a new branch in the repository, locally.
- **git checkout** - Used to switch the current branch.

There are other advanced Git operations, but these are the most basic and commonly used.

Here is an overview of some basic Github terminology:

Forking - When a Github user wants to work on an open source repository, she "forks" that

repository. This means there is a copy of that repository associated with that user's account, to which she can push changes (because she probably won't have push access to the main repository). The user can then use git clone to clone this fork of that repository.

Let's say user calvin@gatech.edu creates a Github repo for InTEL. It will have a URL like git://github.com/**calvin**/InTEL). User beth@gatech.edu forks the InTEL repository. She now has a copy with URL git://github.com/**beth**/InTEL. She can perform all the operations listed above (clone, pull, add, commit, etc.) to this fork, but not to the main repository. She can choose to clone git://github.com/calvin/InTEL, but won't be able to push, as that is not her repository.

Pull Requests - Pull requests are used when someone has a patch to contribute to the repository. A patch submitter makes a pull request to the repository maintainer, asking the maintainer to perform a git pull from the submitter's repository copy, and (if it's a patch that is wanted) merge it with the main repository. For example, if beth@gatech.edu has a new patch or feature to contribute to InTEL, she will send a pull request to calvin@gatech.edu. Calvin can then perform a git pull on Beth's fork and merge the changes with his copy if he wants

Below are some simple instructions for performing basic operations using Github:

Clone a Github repository

1. In Windows, start Git Bash. In Linux or Mac OS X, start a terminal.
2. Type in: git clone git://github.com/path-to-repository.

Commit changes to a Github repository (locally)

1. In Windows, start Git Bash. In Linux or Mac OS X, open Terminal.
2. Navigate to the repository directory
3. Type: git add .
   If you added, deleted files or changed the directory structure, add the -A option:
   git add -A
4. Then enter:
   git commit -m 'commit message'
5. To now push these changes to the remote repository, type:
   git push origin master

Here, "origin" is shorthand for "where I cloned this repository from." Above, "master" is the name of the branch you are pushing to, but this can be any other branch as well.

Setting up NetBeans

The InTEL software was developed using NetBeans (http://netbeans.org/), an IDE that is designed for Java programming and supports a variety of other programming languages. The code is checked in including folders for project metadata, to make getting started as easy as possible. It is strongly recommended to use NetBeans in doing work on this project.

In order to run an exercise, the source code must first be built. This step only needs to happen

once to start with, but if changes are made to the Statics module, then it will need to be rebuilt before those changes can be observed in an exercise.

Building the project

The first project you should open in NetBeans is the folder you used to check out the code. This folder has an Ant script inside it that builds all of the parts of the software in the correct order. When you build this project, you should see output that looks something like this:

```
*********************************
*** InTEL Project Master Build ***
*********************************
getJME:
*** Downloading JavaMonkeyEngine.zip
*** This may take a moment...
Expanding: C:\Projects\InTEL\JavaMonkeyEngine.zip into C:\Projects\InTEL
DEPRECATED - The copyfile task is deprecated.  Use copy instead.
*** Done!
build:
*** BUILDING ALL
***
*** Building BUI
BUI.init:
BUI.deps-jar:
BUI.compile:
Building jar: C:\Projects\InTEL\BUI\dist\BUI.jar
Copy libraries to C:\Projects\InTEL\BUI\dist\lib.
Signing JAR: C:\Projects\InTEL\BUI\dist\BUI.jar to C:\Projects\InTEL\BUI\dist\BUI.jar as mykey

Warning:
The signer certificate has expired.
DEPRECATED - The copyfile task is deprecated.  Use copy instead.
BUI.jar:
*** Building AppletLoader
AppletLoader.init:
Deleting: C:\Projects\InTEL\AppletLoader\build\built-jar.properties
AppletLoader.deps-jar:
Updating property file: C:\Projects\InTEL\AppletLoader\build\built-jar.properties
Compiling 1 source file to C:\Projects\InTEL\AppletLoader\build\classes
C:\Projects\InTEL\AppletLoader\src\edu\gatech\statics\applet\AppletLoader.java:88: warning:
sun.security.util.SecurityConstants is Sun proprietary API and may be removed in a future release
import sun.security.util.SecurityConstants;
C:\Projects\InTEL\AppletLoader\src\edu\gatech\statics\applet\AppletLoader.java:1474: warning:
sun.security.util.SecurityConstants is Sun proprietary API and may be removed in a future release

perms.add(new SocketPermission(host, SecurityConstants.SOCKET_CONNECT_ACCEPT_ACTION));
C:\Projects\InTEL\AppletLoader\src\edu\gatech\statics\applet\AppletLoader.java:1478: warning:
sun.security.util.SecurityConstants is Sun proprietary API and may be removed in a future release
                    perms.add(new FilePermission(path, SecurityConstants.FILE_READ_ACTION));
Note: C:\Projects\InTEL\AppletLoader\src\edu\gatech\statics\applet\AppletLoader.java uses
unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
3 warnings
AppletLoader.compile:
C:\Users\Jayraj\Desktop\json-java\json.jar is a directory or can't be read. Not copying the
libraries.
Not copying the libraries.
Building jar: C:\Projects\InTEL\AppletLoader\dist\AppletLoader.jar
```

```
To run this application from the command line without Ant, try:
java -jar "C:\Projects\InTEL\AppletLoader\dist\AppletLoader.jar"
Signing JAR: C:\Projects\InTEL\AppletLoader\dist\AppletLoader.jar to
C:\Projects\InTEL\AppletLoader\dist\AppletLoader.jar as mykey

Warning:
The signer certificate has expired.
DEPRECATED - The copyfile task is deprecated.  Use copy instead.
AppletLoader.jar:
*** Building Statics
Statics.init:
Statics.deps-jar:
Compiling 150 source files to C:\Projects\InTEL\Statics\build\classes
Note: C:\Projects\InTEL\Statics\src\edu\gatech\statics\modes\truss\zfm\ZeroForceMember.java uses
or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
Statics.compile:
Building jar: C:\Projects\InTEL\Statics\dist\Statics.jar
Copy libraries to C:\Projects\InTEL\Statics\dist\lib.
To run this application from the command line without Ant, try:
java -jar "C:\Projects\InTEL\Statics\dist\Statics.jar"
Signing JAR: C:\Projects\InTEL\Statics\dist\Statics.jar to
C:\Projects\InTEL\Statics\dist\Statics.jar as mykey

Warning:
The signer certificate has expired.
DEPRECATED - The copyfile task is deprecated.  Use copy instead.
DEPRECATED - The copyfile task is deprecated.  Use copy instead.
Statics.jar:
*** Building Exercises:
Archer.init:
Archer.deps-jar:
Archer.compile:
Building jar: C:\Projects\InTEL\exercises\Archer\dist\Archer.jar
Signing JAR: C:\Projects\InTEL\exercises\Archer\dist\Archer.jar to
C:\Projects\InTEL\exercises\Archer\dist\Archer.jar as mykey

Warning:
The signer certificate has expired.
DEPRECATED - The copyfile task is deprecated.  Use copy instead.
Archer.jar:
Awning.init:
Awning.deps-jar:
Awning.compile:
Copy libraries to C:\Projects\InTEL\exercises\Awning\dist\lib.
Building jar: C:\Projects\InTEL\exercises\Awning\dist\Awning.jar
To run this application from the command line without Ant, try:
java -jar "C:\Projects\InTEL\exercises\Awning\dist\Awning.jar"
Signing JAR: C:\Projects\InTEL\exercises\Awning\dist\Awning.jar to
C:\Projects\InTEL\exercises\Awning\dist\Awning.jar as mykey

Warning:
The signer certificate has expired.
DEPRECATED - The copyfile task is deprecated.  Use copy instead.
Awning.jar:
Bicycle.init:
Bicycle.deps-jar:
Bicycle.compile:
```

```
Building jar: C:\Projects\InTEL\exercises\Bicycle\dist\Bicycle.jar
Signing JAR: C:\Projects\InTEL\exercises\Bicycle\dist\Bicycle.jar to
C:\Projects\InTEL\exercises\Bicycle\dist\Bicycle.jar as mykey

Warning:
The signer certificate has expired.
DEPRECATED - The copyfile task is deprecated.  Use copy instead.
Bicycle.jar:
.....
```

The warning messages that appear in this log are normal and can be disregarded. This step extracts the JavaMonkeyEngine code and library from a zip file, and then builds the main components and then all the exercises. The messages regarding the signer certificate are due to the jar signing process, which is needed so that the software can be run as an applet.

Running an Exercise

All exercises are located in the “exercises” folder, and can be opened as normal NetBeans projects. If you open one of these and attempt to run it and get this message:

```
Jul 18, 2011 4:17:16 PM edu.gatech.statics.application.Launcher main
INFO: Georgia Tech Statics
Jul 18, 2011 4:17:16 PM edu.gatech.statics.application.Launcher main
SEVERE: Statics Launcher: Need to specify an exercise!
```

Then you need to right click on the exercise project, select properties, and add some information to the “Run” category of information. Specifically, the following information must be added:

1. Main Class: this must be: “edu.gatech.statics.application.Launcher”
2. Arguments: this must be the name of the exercise class, for instance, “panel.PanelExercise” for the 3D panel problem.
3. Working directory: this must be: “../../Statics”

Technically, this means that the class that is being run to launch an exercise is the Launcher, which is part of the Statics jar. The launcher takes as an argument the name of the class for the exercise, and it must be run within the Statics folder, to have access to the files located there.

If this information is entered correctly, the problem should appear correctly when you run the project.

Special Commands

When you run the InTEL software in standalone mode (outside of an applet), there are several special commands that you can access via the function keys.

- F9: This Saves the current state to the file Save.statics. The contents of this file are the same as what is sent back to the toolkit through auto_postAssignment.php (as discussed earlier). This data can be decrypted through using edu.gatech.statics.util.Base64UI
  saving data is useful for discerning whether persistence is working correctly.

- F10: This loads the state in Save.statics into the current exercise. If the saved state is from a different exercise, this will likely produce many error messages and an invalid state.
- F11, F12: These toggle the visibility of measurements and the UI, respectively. This can be very useful for taking screenshots of problems for presentation purposes.

Folder Structure

This section describes all the folders that are present in the main InTEL directory, and explains the main purposes of each.

applet

This folder is where the binaries are copied. Binaries for modules and exercises (such as Statics.jar, BUI.jar, Bicycle.jar, and so on) are not under version control. Binaries for LWJGL are under version control so that they can be updated when needed. The files .keystore (used for jar signing) and version.json (used for version tracking), are also stored in this folder. This folder is intended to be copied essentially wholecloth into the applet folder in the web.

AppletLoader

This is a Java project that handles the initial loading screen when the user views an applet in a browser. The AppletLoader downloads the jars that need to be downloaded and deals with extracting and loading the native libraries.

BUI

The BUI module is the user interface that is used in the software. This is based off of the “Banana User Interface” (http://samskivert.com/code/jme-bui/) originally developed by Michael Bayne of Three Rings. BUI as it was originally developed is no longer being maintained, so the code that we have contains a number of custom edits to make the interface package more useful for the InTEL software.

The BUI architecture is designed to resemble Swing but also supports stylesheets.

exercises

All the exercises go in this folder.

FlashProblems

The Flash problem “Learning Connectors” has its source located in here.

JavaMonkeyEngine

The Java Monkey Engine is a game engine for Java, using LWJGL as a rendering system. This engine is the underlying platform that supports the Statics module and the entire InTEL project. The version of JME we use is 2.0, though the current version of the JME platform is 3.0 (http://jmonkeyengine.org/). This project began when JME was still in its pre 1.0 period. We have not had the chance or need to update to the latest version, of the engine, and given the number of changes present, it would be a massive and costly undertaking.

manipulatableProblems

This folder contains the Processing projects for the manipulatable problems that are available on the web toolkit: Merry go Round and See-Saw.

nbproject

Contains the metadata for the project that controls the build.xml in this folder.

Statics

This contains the Statics module, the main part for all the InTEL software. The contents of this project are explained in detail later in this document. This folder is also where exercises are run when they are launched as standalone outside of applets. The folder contains binary files for LWJGL, as well as the properties.cfg, which sets up the display configuration.

toolkit

This folder contains all the files that are used for the web based toolkit, which is mostly PHP source code. The contents of this folder are explained in detail later in this document as well.

Versioning

This project builds the version.json file that is located in the applet directory. This is used by the AppletLauncher to ensure that the applet will only download the jar files that are missing or whose versions have changed. Versioning should be run whenever jar files in the applet directory have changed (whenever anything has been rebuilt) and the updated version.json file should be uploaded with the changed files to ensure that clients are getting the correct version of the files.

Viewer

The viewer works just like an ordinary exercise (though its run directory is "../Statics" instead of "../../Statics"), and is designed to load Collada files. The 3D files that are used to represent structures in the Statics exercises must be in the Collada format. The importing process from this format into JavaMonkeyEngine is notoriously finicky and error-prone, so the Viewer is useful for verifying that Collada files are loaded correctly.

Other folders

The folders InterfaceDemo, JMEPhysics, ManipulatableStatics, Maya archive, and Ontology were all created for purposes that are no longer in effect. These can be safely ignored.

Creating an exercise

This section describes the steps necessary to create a new exercise in InTEL. First, it contains instructions on how to create a Netbeans project for the exercise. Next, it will walk through the

steps common to all classes of problems. Finally, the specific details for each class of statics problem will be discussed.

Creating the project

Problems for InTEL have a few differences from ordinary Java projects that one might create in Netbeans. To create a new problem, you should proceed along the following steps:

1. Click on File→New Project. In the Choose Project dialog, select Java Application and click Next.
2. In the Name and Location dialog, enter whatever you want your exercise to be named in Project Name. In the Project Location field, select the exercises folder in the InTEL directory.
3. Uncheck the “Create Main Class” check box as well as “Set as Main Project” and click Finish. (Exercises do not have a main class in the traditional sense.)
4. In the Projects panel in the left of the Netbeans window, create a new package by right-clicking on “Source Packages” and give it the same as your project, but lower-cased. All the Java source files will go into this package.
5. Right-click on this package and create a new Java class. Name it <your_project_name>Exercise.java. This is the project exercise file.
6. Create another package in the same way, named <your_project_name_lower-cased>.assets. All your exercise assets (models, images, etc.) will go into this package.
7. Right-click on Libraries and select Add JAR/Folder. Select the following .jar files: BUI.jar, Statics.jar and JavaMonkeyEngine.jar, lwjgl.jar, lwjgl_util.jar, jme-colladabinding.jar. All of these can be found in the applet directory in the InTEL project folder.
8. Go into the project properties and under the “Run” category and enter the following information: Under Main Class put: “edu.gatech.statics.application.Launcher”, under Arguments put the name of the exercise class including its package (for instance, “panel.PanelExercise” for the 3D panel problem), under Working Directory put: “../../Statics”. This ensures that the exercise will be launched correctly.
9. Finally, to be able to deploy the applet, you must edit the build.xml file to sign and copy the jar into the applet directory. The build.xml file can be accessed within Netbeans by using the Files view instead of the project view and looking within the project folder. Add the following segment to the file after the import for build-impl.xml.

```
<target name="-post-jar">
    <signjar jar="dist/Levee.jar" alias="mykey" storepass="monkeys"
             keystore="../../applet/.keystore"/>
    <copyfile src="dist/Levee.jar" dest="../../applet/Levee.jar" />
</target>
```

However, replace “Levee” with the name of your exercise.

Types of exercises

Any problem that can be run by InTEL must be a subclass of Exercise. Exercise on its own is an abstract class, and is not usually meant to be subclassed on its own. There are several existing subclasses of Exercise that provide support for different types of statics problems. This subsection will acquaint you with the different kinds of exercises, and details of how to build exercises of these types will be explained later in this section.

- **OrdinaryExercise**
  This is an exercise that supports solving for equilibrium in one or more bodies.

- **SimpleFBDExercise**
  The simple FBD exercise does not enable the equation mode, so if you wish to create a problem that only teaches free body diagrams.

- **TrussExercise**
  The truss exercise is designed for problems that consist exclusively of PointBodies and TwoForceMembers. The truss exercise makes the user identify zero force members before proceeding in the diagram. There is also a special mode associated with creating section diagrams.

- **FrameExercise**
  This is closely related to OrdinaryExercise, but designed for frame problems, and allows the user to easily select all members.

- **CentroidExercise**
  The centroid exercise is one of the more unusual types of exercises, allowing students to solve for the centroid of a body. This does permit FBD and equation modes, but these have not been used in any existing centroid problem so far.

- **Ordinary3DExercise**
  This exercise has modifications to enable OrdinaryExercises that take place in 3D. This is primarily a change to UI elements, giving 3D navigation controls and updating the equations.

- **DistributedExercise**
  This enables support for distributed loads. This works by enabling a new mode to solve for the resultant of the distributed force.

It is important to note that all of these are simply subclasses of Exercise, and they differ only in how they treat the interface and the different modes that are enabled for the problem. If you need to create a hybrid type of exercise, such as a problem that has a frame but also uses distributed loads, this is possible, but must be done by combining UI elements of both exercise classes.

Writing the Exercise class

The exercise file should contain one class named <Your_Project_Name>Exercise. It should extend one of the base exercise classes described above.

Problem Description

The first part of the exercise file is the problem description. This can be set by overriding the getDescription() method to set the problem title, narrative, problem statement and goals and also add images.

```
public class SeeSawExercise extends SimpleFBDExercise {//OrdinaryExercise {

    @Override
    public Description getDescription() {
        Description description = new Description();

        description.setTitle("See Saw");
        description.setNarrative(
                "Sam is at the park with his nephew and niece, Jenny and Bobby, " +
                "who are balancing on a see-saw.  He is taking a statics class at " +
                "Georgia Tech and has recently learned about free body diagrams.  " +
                "Can you help him make a FBD of the see saw?");

        description.setProblemStatement(
                "The see saw is supported with a pin at point B.  " +
                "Jenny and Bobby are located at points A and C respectively.");

        description.setGoals("Build a free body diagram of the see saw.");

        description.addImage("seesaw/assets/seesaw-0.png");
        description.addImage("seesaw/assets/seesaw-1.jpg");
        description.addImage("seesaw/assets/seesaw-2.jpg");
        description.addImage("seesaw/assets/seesaw-3.jpg");

        return description;
    }
```

If the problem description exceeds the available space, it is possible to make sure that the description page has a scrollbar.

```
description.setLayout(new ScrollbarLayout());
```

Initialization

The second step is to initialize the exercise by overriding the initExercise() method. This involves setting the units of measure for the problem. The available units that can be adjusted are:

- distance
- angle
- force

- forceOverDistance
- moment
- mass
- specificWeight (this was used only once in the Levee exercise)
- none (this is for unit-less terms, such as friction coefficients)

The line including “Unit.setDisplayScale(...)” describes how the world units should relate to the units displayed in measurements. This should only be used for distances. The number used here should be a power of ten. For instance, if there are two points that are 3 units apart in world coordinates, then a display scale of 10 and a distance of meters would mean those points are 0.3 meters apart. If, on the other hand, the scale is 0.1, then the points would be 30 meters apart.

Using the display scale is a slightly awkward way of scaling problems, but is useful for having a range of numeric values without scaling the coordinates used by the 3D engine.

```
@Override
public void initExercise() {

    Unit.setSuffix(Unit.distance, "m");
    Unit.setSuffix(Unit.angle, "");
    Unit.setSuffix(Unit.force, "N");
    Unit.setDisplayScale(Unit.distance, new BigDecimal("10"));

    getDisplayConstants().setForceSize(1);
    getDisplayConstants().setMomentSize(1);
    getDisplayConstants().setPointSize(1);
    getDisplayConstants().setMeasurementBarSize(1f);

}
```

Each type of unit can have a precision (in this case, meaning the number of digits displayed after a decimal point):

```
Unit.setPrecision(Unit.moment, 2);
Unit.setPrecision(Unit.force, 2);
Unit.setPrecision(Unit.mass, 2);
Unit.setPrecision(Unit.forceOverDistance, 2);
```

Interface Configuration

Most of the time, you will not need to touch the method createInterfaceConfiguration. The InTEL software supports the ability to plug in and swap interface elements in different problems. The interface configuration (IC) is responsible for almost all the UI elements that appear in the InTEL software. Specifically, the IC controls the mode panels, the navigation window, the view constraints, and the knowns and tasks windows.

The IC tends to be sufficiently configured for most types of problems. Occasionally, it will be

necessary to change the type of navigation window (for instance from 2D to 3D navigation), or adjust the view constraints.

```
@Override
public AbstractInterfaceConfiguration createInterfaceConfiguration() {

    AbstractInterfaceConfiguration interfaceConfiguration = (AbstractInterfaceConfiguration) super.createInterfaceConfiguration();
    interfaceConfiguration.setNavigationWindow(new Navigation3DWindow());

    ViewConstraints vc = new ViewConstraints();
    vc.setPositionConstraints(-1, 1, -1, 10);
    vc.setZoomConstraints(0.5f, 4f);
    vc.setRotationConstraints(-1, 1, -1, 1);
    interfaceConfiguration.setViewConstraints(vc);

    return interfaceConfiguration;

}
```

The view constraints control how much freedom the camera has in panning, zooming, and rotating around the problem space.

The IC is designed to be extensible in case additional UI elements are needed in new types of problems.

Load Exercise

The loadExercise() method is where the actual problem is constructed.

The exercise creation process, based on the information from the problem document, is performed within this method. The objects and the order of objects used in this method depend on the concepts involved in the particular exercise. Even though the steps within this method might vary with every problem, there are a few main ideas employed in the creation of all exercises.

The ideas differ technically in 2 ways. Some of the objects used are involved in the actual creation of components and concepts behind the problem and the other objects are used to display the previous former objects on to the screen. The most streamlined method would be to implement both these ideas together at every step of the exercise creation process.

```
Schematic schematic = getSchematic();

Point A = new Point("A", "0", "0", "0");
Point B = new Point("B", "0", "5", "0");
Point C = new Point("C", "0", "10", "0");
Point D = new Point("D", "4", "10", "0");
Point E = new Point("E", "4", "5", "0");
Point F = new Point("F", "4", "0", "0");

A.createDefaultSchematicRepresentation();
B.createDefaultSchematicRepresentation();
C.createDefaultSchematicRepresentation();
D.createDefaultSchematicRepresentation();
E.createDefaultSchematicRepresentation();
F.createDefaultSchematicRepresentation();

schematic.add(A);
schematic.add(B);
schematic.add(C);
schematic.add(D);
schematic.add(E);
schematic.add(F);
```

A basic example of the mentioned methodology is shown in the above snippet of code. The points are first created according to their spatial arrangements described in the problem statement. A default schematic creation is created for each of the points so they can be added to the overall **schematic** of the display.

```
Bar BE = new Bar("BE", B, E);
Beam ABC = new Beam("AC", A, C);
Beam CD = new Beam("CD", C, D);
Beam DEF = new Beam("DF", D, F);
```

This method of creating the component objects first and then displaying it using the **schematic** object is followed throughout the exercise creation process. It can be used to create connectors, bars, beams etc and also to generate relationships between the objects (points) and their connections (connectors) to set up the concepts behind the exercise (such as Tension between two points).

```
//adding distributed forces for beam CD
DistributedForc                                                    orce", CD
new Vector(Unit                                                    )));
DistributedForceObject distributedtrussObject = new DistributedForceObject(distributedtru
//Added by Jayraj to fix no name problem
//distributedtrussObject.setName("trussForce");
CD.addObject(distributedtrussObject);

//arrow graphic size and create schematic representation of arrows (adding to visual scen
int arrowDensity = 2;
distributedtrussObject.createDefaultSchematicRepresentation(18f / 6, 2 * arrowDensity, 1.

//creating distance measurement between points
DistanceMeasurement measureFull = new DistanceMeasurement(A, F);
measureFull.createDefaultSchematicRepresentation();
measureFull.setName("Measure AF");
schematic.add(measureFull);

//Adding pins to end points
Pin2d endA = new Pin2d(A);
endA.attachToWorld(ABC);
endA.setName("support A");
endA.createDefaultSchematicRepresentation();

Roller2d endF = new Roller2d(F);
endF.setDirection(Vector3bd.UNIT_Y);
endF.attachToWorld(DEF);
endF.setName("Support F");
endF.createDefaultSchematicRepresentation();

Connector2ForceMember2d connectorB = new Connector2ForceMember2d(B, BE);
connectorB.attach(ABC, BE);

Connector2ForceMember2d connectorE = new Connector2ForceMember2d(E, BE);
connectorE.attach(DEF, BE);

Fix2d fixC = new Fix2d(C);
fixC.setName("fix C");
fixC.attach(ABC, CD);
Fix2d fixD = new Fix2d(D);
fixD.setName("fix D");
fixD.attach(DEF, CD);

schematic.add(BE);
schematic.add(ABC);
schematic.add(CD);
schematic.add(DEF);
```

public class edu.gatech.statics.modes.distributed.objects.DistributedForceObject

Once the points, connectors and the relationship between them are set up according to the underlying problem idea, these components are then attached to the 3D Model to be displayed. Although, this is the last step in the exercise creation process, it would be a good idea to work on this along with the creation of points so as to constantly monitor all the components that are being displayed.

```
public class edu.gatech.statics.objects.representations.ModelNode
ModelNode modelNode = ModelNode.load("centergyframe/assets/", "centergyframe/assets/CenturgyB_collada.
modelNode.extractLights();

//Specifying the root folder for all the model objects that need to be attached to the exercise elemen
ModelRepresentation rep;
String prefix = "VisualSceneNode/everything/everything_group1/Scene/";

//Attaching a model object to it's respective exercise element
rep = modelNode.extractElement(ABC, prefix+"pCube3");
ABC.addRepresentation(rep);
rep.setSynchronizeRotation(false);
rep.setSynchronizeTranslation(false);

rep = modelNode.extractElement(CD, prefix + "pCube16");
CD.addRepresentation(rep);
rep.setSynchronizeRotation(false);
rep.setSynchronizeTranslation(false);

rep = modelNode.extractElement(DEF, prefix + "pCube2");
DEF.addRepresentation(rep);
rep.setSynchronizeRotation(false);
rep.setSynchronizeTranslation(false);

rep = modelNode.extractElement(BE, prefix + "pCube27");
BE.addRepresentation(rep);
rep.setSynchronizeRotation(false);
rep.setSynchronizeTranslation(false);

rep = modelNode.getRemainder(schematic.getBackground());
schematic.getBackground().addRepresentation(rep);
```

This can be done before setting up the relationship between the points and connectors – as per the problem document. Tasks are added at the end to ensure clear solution requirements for the student and to instigate a prompt once the problem has been completed.

In order for a problem to be solvable, it needs to have tasks added to it.

```
//Adding Tasks
addTask(new SolveConnectorTask("Solve A", endA));
addTask(new SolveConnectorTask("Solve F", endF));
```

There are a few types of tasks that students can complete in problems:

- **SolveConnectorTask** - this takes a connector and is complete when that connector is marked as solved.
- **Solve2FMTask** - this takes a 2fm and a connector that is attached to it. When the 2fm is solved, this reports whether the member is in tension or compression.
- **CompleteFBDTask** - this takes a BodySubset and reports solved whenever the FBD for this particular body subset is solved. This is useful for SimpleFBDExercises.
- **SolveFBDTask** - this takes a BodySubset and reports solved whenever the FBD and equations for this particular body subset are solved.
- **SolveCentroidBodyTask** - this takes a CentroidBody and reports solved whenever the body has its centroid found.
- **SolveZFMTask** - this reports solved when the zero force members in a truss problem are found.

Constructing a problem

Developing an exercise can be a daunting process even if you have a solid understanding of both Java and Statics. While there is a lot to keep track of, it is possible to separate every exercise into several simple pieces.

The process of developing a new exercise involves constructing all the objects representing the pieces of information needed for solving the problem. These pieces can then be connected together and added to the `Schematic`. All of the objects that are added to the `Schematic` are subclasses of `SimulationObject`. Any part of an exercise that can be displayed and interacted with by the user is a `SimulationObject`. Examples of these are: `Point`, `Body`, `Connector`, `Load`, and `Measurement`.

`SimulationObjects` can have `Representations`, which allow them to be displayed. Each subclass of `SimulationObject` must implement a method called `createDefaultSchematicRepresentation`, which creates something that will allow the object to be displayed when the user opens up a problem. Without a `Representation`, a `SimulationObject` is invisible. Representations may be customized and re-assigned. In problems where there are 3D models, `ModelRepresentations` are attached to Bodies in the problem.

Every `SimulationObject` must have a unique name. The name is typically the first argument in the constructor. If a `SimulationObject` does *not* have a unique name, the exercise will throw an exception before it starts up.

Points

The very first part of developing an exercise is to make a list of the points that are needed in the problem. Most problems will have measurements that help the user identify where the points are located in space. Points are typically constructed like this:

```
Point A = new Point("A", "0", "1", "2");
```

The first argument in the constructor is the name, which is unique. The user will see this name on the label if they use the default representation. The rest of the arguments are the coordinates of the point. These are given as `String`s, because `Point` uses `BigDecimal` to store its underlying coordinates. Floating point numbers cannot represent numbers like 0.1 without rounding, and this occasionally causes problems for users, so all significant values in an exercise are stored using `BigDecimal`s, which are most easily constructed with `String`s.

A few special cases treat points slightly differently:
1) In exercises that have distributed loads, there are points that are “discovered” by the user in the `DistributedMode`. These are not added in the construction of the exercise, but are created by the `DistributedForceObjects` when they are solved.

2) 3D exercises require the user to go through each point and actually enter their coordinates.

3) There is a class, `UnknownPoint`, which was introduced to handle situations where a point could have unknown coordinates that were controlled by an unknown. This wound up not working effectively, so the class is not currently used.

Bodies

Bodies are the central object of study in Statics. Most statics problems involve the isolation of a body or a set of bodies in a schematic via free body diagrams. In the software, a `Body` is a `SimulationObject` which can have other `SimulationObject`s (namely `Load`s and `Connector`s) attached to it.

There are several important subclasses of Body:
1) Potato: The most basic type of Body is a `Potato` (a term is in honor of one of the project PI's, Larry Jacobs). Potato has no special constraints at all, and can be used to describe a body that has an amorphous shape (e.g. human characters, as in the Spiderwoman problem).

2) Beam: Many bodies can be simplified as Beams. A beam is a long object that has two Points as ends. This and other long bodies have a default representation of a gray bar that lies between the endpoints.

3) TwoForceMember: there are two kinds of TwoForceMembers, Bars and Cables. A two-force member is a special kind of long body that can only exert force in between its endpoints. These can also only use a special kind of Connector: Connector2ForceMember2d.

4) ZeroForceMember: this class is derived from TwoForceMember, but it is a special case where the member cannot carry any force at all. These can only be used in Truss problems. It always appears grayed out, and it can not have any connectors attached to it.

5) PointBody: this is a body that is simplified to be observed at only a single point. These are what would be referred to as joints in Truss or Frame problems. Typically, a PointBody is attached only to two-force members, however it is not restrained in what kinds of connections it can have. A point body may be connected to the world by any type of connector.

6) Background: The Background is a subclass of Body, and can be thought of as representing the rest of the world in a problem. The background is there only for the sake of holding representations for the parts of a scene in an exercise that are not supposed to be bound to any other object. There is only one instance of Background, and it is owned by the Schematic. Please don't add connectors or any other objects to the Background.

Connectors

Connectors describe how all the the bodies are attached to each other and the world. All connectors start out as unknown, and the user must solve for any reactions they need in order

to solve the problem. The software supports all the kinds of connectors that are used in statics problems.

There are a few types of connectors (for use in 3D problems) that are not currently implemented, such as hinges, 3D pins, and 3D fixes, but these can be developed and added to the software if need be.

The suffix of “2d” is erroneous in Roller2d and Connector2ForceMember2d. These connectors work regardless of whether the problem is 2d or 3D. The naming convention is part of legacy code and has not been corrected.

All connectors must be created at a `Point` and then attached to one or two bodies. If the connector is attached to one body only, this is done through `attachToWorld`, otherwise you must call `attach`.

There are several main types of connectors:
1) Pin2d, Fix2d, Roller2d
These are the connectors used most commonly in statics problems. Pins and fixes do not require any extra information added to them. A roller must be given a direction in which movement is restricted. This direction can not be reversed, when the user solves for the reactions on a roller, they must place the force in the correct direction. Pins and fixes allow reaction forces to be reversed, however.

2) Connector2ForceMember2d
Two force members have special kinds of connectors that allow them to interact with a problem. The Connector2ForceMember2d is constructed using a Point and a TwoForceMember. The direction and type of the TwoForceMember defines the direction that the reaction force must be added in, and whether or not this force can be reversed.

3) ContactPoint
Friction problems allow the use of connectors that are points of contact. A ContactPoint must be constructed with the Point at which the contact is applied, and the ConstantObject that denotes the coefficient of friction. After the ContactPoint is created, it is necessary to set the direction of both the friction and normal forces. The software does not check to make sure these are perpindicular, and the user must add reaction forces in the directions specified, as they cannot be reversed.

Loads

Loads such as Forces and Moments can be added to bodies in constructing an exercise, and these describe the given loads. Givens can be either known or unknown. Given loads must have names, just like all other simulationObjects, even if they are created with a symbolic name.

In building a problem, you can construct a Force in the following ways:
This line creates an unknown given at A.

```
Force fA = new Force(A, Vector3bd.UNIT_Y, “fA”);
```

This line creates a known given at B:

```
Force fB = new Force(B, Vector3bd.UNIT_Y, new BigDecimal(“1”));
```

Measurements

One of the important and easily forgotten parts of creating an exercise is to incorporate measurements. Without these, it is impossible for the student to ascertain where points are in relation to each other. The two main kinds of measurements are distances and angles. Measurements may also be attached to bodies to ensure that they stay with the body through various diagrams.

DistanceMeasurements are created between two points. Calling forceVertical or forceHorizontal restricts the way the measurement is displayed so that it displays only the distance along the Y or the X axis, respectively. For instance, a DistanceMeasurement between points located at (0,0,0) and (10,5,0) with forceVertical applied would show a vertical line that had a value of 5 units.

AngleMeasurements can be created as PointAngleMeasurements or FixedAngleMeasurements. PointAngleMeasurements are given three parameters, the first of which is the corner of the angle. So, PointAngleMeasurement(B,A,C) will measure the angle ABC. A FixedAngleMeasurement takes its first two arguments as points, and the third argument is a Vector3f. FixedAngleMeasurement(A,B,v) measures the angle between the lines AB and v.

For both DistanceMeasurement and AngleMeasurement, the call to createDefaultSchematicRepresentation can be supplied with an optional parameter indicating its offset, meaning how far away the line should appear from the points being measured. This helps legibility when multiple distances must be shown in the same space.

A final type of measurement which was added in the development of 3D problems and has been somewhat underused is the CoordinateSystem, whose constructor has a single argument denoting whether the measurement should be shown in 3D. This creates a coordinate system situated at the origin indicating the directions of the X and Y axes, as well as the Z axis if the problem is 3D.

Other types of exercises

There are a variety of types of exercises that aim to cover the entire Statics syllabus. It is strongly recommended to use existing exercises as a guide for constructing new problems.

Distributed Force

Examples: Levee, Awning, Bookshelf

A distributed force problem should inherit DistributedForceExercise. Distributed forces should be applied to beams. The beam itself must be constructed between two points.

```
Point A = new Point("A", "0", "0", "0");
Point B = new Point("B", "10", "0", "0");

Point mid1 = new Point("mid1", "2", "0", "0");
Point mid2 = new Point("mid2", "6", "0", "0");
Point mid3 = new Point("mid3", "7", "0", "0");
```

In this problem, the beam is created across points A and B. The points mid1, mid2, and mid3 are used to construct the distributed forces.

```
Beam bookshelf = new Beam("Bookshelf", A, B);
```

The points and beam can then be used to create a DistributedForce:

```
DistributedForce dlBooks1 = new ConstantDistributedForce("books1", bookshelf, A, mid1,
        new Vector(Unit.forceOverDistance, Vector3bd.UNIT_Y.negate(), new BigDecimal("0.20")));
DistributedForceObject dlObjectBooks1 = new DistributedForceObject(dlBooks1, "1");
dlObjectBooks1.setName("Force for Book Group 1");
bookshelf.addObject(dlObjectBooks1);
```

You must specify the direction of the force in the DistributedForce. The numeric parameter 0.20 that is passed in this example denotes the value of the maximum force/distance for this particular distribution. We use a DistributedForce to make a DistributedForceObject. The DistributedForce describes the mathematics and numerical values, while the DistributedForceObject is the object that appears and can be manipulated in the software. The DistributedForceObject must have a name, and must also be added to the beam onto which it is applied.

Centroid

Example: Space Station

The goal of working a centroid exercise is to find out the centroid of a CentroidBody which has a complex shape. Centroid exercises should extend the class CentroidExercise. The shape of the CentroidBody is defined by a number of CentroidParts. Currently the only type of CentroidPart that is used is a RectangleCentroidPart, but there is programmatic support for additional types (such as discs and triangles). Much like the distributed exercises, the CentroidParts must have CentroidPartObjects constructed using them in order to be accessible in the problem. These CentroidPartObjects should get added to the CentroidBody.

When a student goes to solve a centroid exercise, they can select the CentroidBody and then solve the centroids of its individual parts before solving for the centroid of the whole thing.

It is important to make sure there are sufficient measurements present in the schematic to allow

the student to correctly make sense of the problem.

Friction

Examples: Keyboard (friction), Spiderwoman, LadderDrill

Friction exercises do not actually require a special problem type, so it is possible to have an OrdinaryExercise, or even a truss or frame exercise with friction. The only thing that is necessary to add friction to an exercise is to correctly set up a ContactPoint. Friction occurs when two bodies are in contact, and the bodies exert force on each other at that point.

A contact point needs to be created using a ConstantObject, which represents the value for its coefficient of friction. This needs to be created and added to the diagram:

```
frictionObjectB = new ConstantObject("mu B", Unit.none);
schematic.add(frictionObjectB);
```

Next, the points are created for the specific coordinates according to the problem diagram.

```
A = new Point("A", "-5.5", "11.2", "0");
B = new Point("B", "0", "0", "0");
G = new Point("G", "-6", "5.5", "0");
```

The ContactPoint works just like any other connector, but it requires its constant object to be supplied in its constructor, and also requires you to explicitly set the direction of the normal and friction forces.

```
jointB = new ContactPoint(B, frictionObjectB);
jointB.setName("ContactB");
jointB.setNormalDirection(Vector3bd.UNIT_X.negate());
jointB.setFrictionDirection(Vector3bd.UNIT_Y);
```

Afterward, the ContactPoint can attach the body to the world or another object just like any other connector.

```
body = new Potato("Spiderwoman");
jointB.attachToWorld(body);
```

Frame/Truss

Frame examples: Arm and Purse, Bicycle, Keyboard, CRC Roof
Truss examples: Roof, Bridge

Caveat: The Bridge problem is *atypical*. Because the problem is so big, it was constructed in an unusual way: algorithmically instead of manually. Given the complexity of the bridge, you should use the Roof problem as a guide for constructing new truss problems.

The rest of this section reviews how the bicycle problem has been constructed. The first step is the creation of points:

```
A = new Point("A", "0", "0", "0");
B = new Point("B", "3", "5.196", "0");
C = new Point("C", "9", "5.196", "0");
D = new Point("D", "12", "0", "0");
E = new Point("E", "6", "0", "0");
F = new Point("F", "4", "6.928", "0");
G = new Point("G", "1.5", "2.598", "0");
```

The bicycle is created with one beam, several bars, and three point bodies. PointBodies represent what are called joints in statics. These are bodies that have several forces applied to them via two force members and other connectors.

```
FA_handlebarBeam = new Beam("Beam FBGA", F, A);
CE_seatPoleBar = new Bar("Bar CE", C, E);
CB_topBar = new Bar("Bar CB", C, B);
CD_backBar = new Bar("Bar CD", C, D);
DE_bottomBar = new Bar("Bar DE", D, E);
GE_frontBar = new Bar("Bar GE", G, E);
BE_middleBar = new Bar("Bar BE", B, E);
jointAtD = new PointBody("Joint D", D);
jointAtC = new PointBody("Joint C", C);
jointAtE = new PointBody("Joint E", E);
```

The Bicycle has several given forces added to it that operate on the various bodies in the problem.

```
Force seatWeight = new Force(C, Vector3bd.UNIT_Y.negate(), new BigDecimal(500));
seatWeight.setName("Seat Y");
jointAtC.addObject(seatWeight);

Force pedalWeight = new Force(E, Vector3bd.UNIT_Y.negate(), new BigDecimal(150));
pedalWeight.setName("Pedals");
jointAtE.addObject(pedalWeight);

Force wedgeForce = new Force(C, Vector3bd.UNIT_X, new BigDecimal(17.3205));
wedgeForce.setName("Seat X");
jointAtC.addObject(wedgeForce);
```

Each TwoForceMember requires a Connector2ForceMember2d at each end where it attaches to another body. Most of these connections are between the various PointBodies, and some connect to the beam (FA_handlebarBeam) that makes up the bicycle's front.

```
// *** all connectors that attach to C
Connector2ForceMember2d c2fm_CB_C = new Connector2ForceMember2d(C, CB_topBar);
Connector2ForceMember2d c2fm_CE_C = new Connector2ForceMember2d(C, CE_seatPoleBar);
Connector2ForceMember2d c2fm_CD_C = new Connector2ForceMember2d(C, CD_backBar);
c2fm_CB_C.attach(CB_topBar, jointAtC);
c2fm_CE_C.attach(CE_seatPoleBar, jointAtC);
c2fm_CD_C.attach(CD_backBar, jointAtC);

// *** all connectors that attach to E
Connector2ForceMember2d c2fm_GE_E = new Connector2ForceMember2d(E, GE_frontBar);
Connector2ForceMember2d c2fm_BE_E = new Connector2ForceMember2d(E, BE_middleBar);
Connector2ForceMember2d c2fm_CE_E = new Connector2ForceMember2d(E, CE_seatPoleBar);
Connector2ForceMember2d c2fm_DE_E = new Connector2ForceMember2d(E, DE_bottomBar);
c2fm_BE_E.attach(BE_middleBar, jointAtE);
c2fm_GE_E.attach(GE_frontBar, jointAtE);
c2fm_CE_E.attach(CE_seatPoleBar, jointAtE);
c2fm_DE_E.attach(DE_bottomBar, jointAtE);

// *** all connectors that attach to D
Connector2ForceMember2d c2fm_DE_D = new Connector2ForceMember2d(D, DE_bottomBar);
Connector2ForceMember2d c2fm_CD_D = new Connector2ForceMember2d(D, CD_backBar);
c2fm_DE_D.attach(DE_bottomBar, jointAtD);
c2fm_CD_D.attach(CD_backBar, jointAtD);
```

The bicycle connects the frame to the ground via two rollers (for each wheel). This means that the bicycle is not actually fully constrained, though it is in static equilibrium. Frequently truss and frame problems will attach the structure to the world via a pin and a roller, or a single fix, allowing for three unknowns.

```
rollerA = new Roller2d(A);
rollerA.attachToWorld(FA_handlebarBeam);
rollerA.setDirection(Vector3bd.UNIT_Y);
rollerA.setName("Roller A");
```

3D

Example: Panel
3D exercises should extend Ordinary3DExercise. Other than having points situated along all three axes, 3D exercises should create a CoordinateSystem object, which is necessary for students to recognize the alignment of the axes and the origin. It is especially important to have adequate measurements so that the student can calculate all the points.

Software architecture

This section describes some of the core elements used in the Statics module of the InTEL software and should be useful to programmers who are trying to extend, debug, or understand the software better.

Key classes

This subsection describes a few of the key classes that are used in the software and how they fit together.

StaticsApplet and AppletLauncher

AppletLauncher is the actual class that holds the applet when a student looks at a problem online. The big thing that this does is to create the StaticsApplication and load the exercise. This also configures the important parts that are used by JavaMonkeyEngine and LWJGL in applets (though this occurs in BaseStaticsApplet). All main functionality such as initialization, rendering, shutting down, and state updates are delegated to StaticsApplication.

StaticsApplication

This class handles a variety of central underlying tasks. There is a global singleton instance of StaticsApplication (which is cleared out at shutdown), accessible by StaticsApplication.getApp(). Practically, it is this class that allows modes to operate in the abstract manner that they do.

StaticsApplication holds a reference to the current diagram and exercise, as well as loggers and listeners. Interacting with StaticsApplication is the best way to provide the user feedback. This class contains functionality that is important throughout the software but does not necessarily belong anywhere else.

Generally, it should not be necessary to work within the StaticsApplication class, however it is useful to engage with many of its public methods, particularly concerning feedback, tools, and how diagrams are changed. It is best to understand these through how calls are already made in existing code.

InterfaceRoot

InterfaceRoot interacts with BUI much the same way that StaticsApplication interacts with the JavaMonkeyEngine. This is the class that handles the windowing system, the view (specifically the camera), and all other tabs and UI elements except for labels. An instance of InterfaceConfiguration from the Exercise is used to customize which windows appear and what are the available mode panels.

Generally, it should not be necessary to interact with InterfaceRoot directly, unless you are performing some very significant change to the interface. Otherwise, all interface customization can occur through InterfaceConfiguration.

Exercise and ExerciseState

The base class for creating new problems is Exercise. Exercise itself has a fair bit of important functionality already built inside of it. Since there can only be one exercise at a time, there is a singleton instance of the class, which you can access via Exercise.getExercise().

The methods that new *types* of exercise must implement or override are supportsType and createNewDiagramImpl, which handles the actual creation of new diagrams according to their types. A useful way to understand how these methods are used is to compare the subclases OrdinaryExercise with DistributedExercise or TrussExercise.

Exercise also contains a reference to its ExerciseState. As a State object, this class is meant

to contain only the save-able data behind what is going on in the exercise. The exercise state contains information about the user and the assignment, any parameters that the problem might have, the SymbolManager, and all diagrams that currently exist.

The symbol manager tracks all of the symbolic quantities that have been either named or solved for. Symbolic quantities are usually Loads, but they can also be ConstantObjects (such as in friction problems). Early in implementation, we intended to have measurements (distances and angles) that could have symbolic quantities, but this never panned out.

Diagram and DiagramState

The main view of the InTEL software always displays a Diagram. Everything that has a tab must have a diagram. The Diagram class is intended to represent the functional view and interaction around a specific type of work, determined by the Mode.

Most diagrams have two defining properties: Their DiagramType (defined by the Mode) and their DiagramKey. The key indicates what part of the problem is being represented by the current diagram. Some diagrams, such as the SelectDiagram or the ZFMDiagram do not have a key, because they address the whole problem. The FreeBodyDiagram and DistributedDiagram have keys of BodySubset and DistributedForce, respectively. Because these keys are used to index diagrams, they must correctly implement equals() and hashCode() so that the appropriate diagram can always be found.

Some diagrams can replace others. For instance, the EquationDiagram will replace a FreeBodyDiagram after the FBD is solved. The FreeBodyDiagram object still exists (and continues to be used by the EquationDiagram), but the EquationDiagram is displayed instead and uses the same tab. This works in code because of the getRecentDiagram() method in Exercise.

Diagram maintains a current DiagramState, which separates all its important state data. DiagramState is subclassed but none of its subclasses should be mutable. This is very important: **subclasses of DiagramState should not be mutable**. Specifically, it should not be possible to ever make changes to the state directly. Instead, new states should be created (either through Builders or through Actions), and these states are pushed onto the Diagram's StateStack. This is how undo and redo are implemented, by performing pushing and popping on this stack.

Diagram is responsible for maintaining all SimulationObjects that are currently visible, and identifying which of these can be interacted with by the user (via the SelectionFilter).

Frequently diagrams can be subclassed to adjust their functionality. For instance, the class SelectionDiagram is subclassed as FrameSelectDiagram in order to prevent the selection of TwoForceMembers and also enable the button to select all members. The Exercise class is responsible for constructing new diagrams, and this is how these subclasses can be instantiated.

Representation

Representation is one of the messier classes, so it will be useful to review a little bit about what it does. Every representation holds a reference to the SimulationObject it represents, and a JavaMonkeyEngine Node object, that contains the actual representation. Forces, moments, and points have representations that use simple polygon primitives, but the class ModelRepresentation can use parts of 3D models.

The body of the Representation class uses coloring and lighting to handle highlighting. There are four types of display states: normal, grayed, hover, and selected. These each have separate types of colors and lighting that can be used for each of these states. It is possible for exercises to manually adjust the color or lighting values on individual representations for different display states.

Serialization and Persistence

The package edu.gatech.statics.exercise.persistence contains the classes that are used for the persistence system. This system is how it is possible to save and restore the user's state data. Persistence works by using XML Serialization, discussed in several articles (http://java.sys-con.com/node/37550, http://java.sun.com/products/jfc/tsc/articles/persistence4/).

The serialized states that are entered into the database and saved into Save.statics are the result of generating an XML output, applying zip compression, and then transforming that data to Base64 format (which can be represented in UTF-8). This encoding and decoding takes place in StateIO. You can use Base64UI to decode these files to examine the actual XML formatting.

At its base level, XML Serialization works by using reflection to identify get and set methods of a class, and creating what is essentially instructions for how to construct a new object that is equivalent to an old one. This process can be customized using PersistenceDelegates, which eliminates the need for public accessors for every single property of a serialized class. However, it is still necessary for persistable objects to have a public zero-argument constructor. Many SimulationObjects have constructors like these that are marked as @Deprecated for this reason. If there is not a zero argument constructor, then the serializer will generate an exception that can be found in the log, and will also prevent the state from saving correctly.

DiagramStates are encoded wholecloth, and so every object that the diagram state has a reference to must be able to be serialized. The StaticsXMLEncoder class defines persistence delegates for many classes, and if you create a new type of mode, then it is likely you will need to add delegates here.

Many objects are given the namedPersistenceDelegate, which saves the object's name only. When state is restored, these objects are not actually restored directly, but the StaticsXMLDecoder instead looks the objects up from the exercise schematic. For instance, if

the serializer is attempting to write a Force, then it will save the force's name, value, and the Point at which the Force is located. That point will be saved using the named delegate, and when the force is restored in a new instance of the program, the decoder will find the point in the exercise and attach the force to that point instead of constructing a new one.

The ability to use existing objects on serial restoration is not something originally supported by the Java XML serialization specification. Furthermore, the relevant classes and methods that would need to be overridden were package protected in the persistence library. Our implementation worked around this by copying several of these classes entirely into the edu.gatech.statics.exercise.persistence package in order to set up the required functionality. Anything in there that is by "Philip Milne" or "Sergey A. Malenkov" is one of these copied classes.

Mode System

The mode system is how InTEL is able to support so many different types of Statics problems. Each type of user activity in problem solving is split into its own mode. The modes that we currently have are as follows:

- DescriptionMode
- SelectMode
- FBDMode
- EquationMode
- ZFMMode (Trusses)
- TrussSectionMode (Trusses)
- FindPointsMode (3D)
- DistributedMode
- CentroidMode

The organization of the modes should be fairly evident from the package tree. Mode packages tend to have subpackages for UI, objects, and actions. This is not mandatory, but has proven useful for organizing data structures. The distributed package is a fairly clear example of how these elements fit together.

Each mode has a set of related classes that are necessary for the mode to do its job.

Mode

The mode class does not do very much on its own, but works as a global indicator representing this mode's functionality. Each Mode subclass should hold a static constant for its instance. Subclasses also must implement createDiagramType to create a unique identifier that can be used to index diagrams.

Mode contains a little bit of functionality that is important for taking care of transitions between diagrams. Mode.load() is the method that should be called whenever a new diagram should be loaded. Mode also has overridable methods preLoad and postLoad, which can be used to

handle any important setup that needs to happen.

Diagram

The Diagram class has already been discussed earlier. Creating a new mode involves creating the associated diagram and its state. The key behind developing these classes is to identify the activity that the user must perform (which takes place in Diagram) and the data that is saved as a result of that activity (the DiagramState).

In order for the diagram to be accessible, it needs to be constructable through the Exercise class. This frequently involves developing a new Exercise class for the particular type of activity that is being defined (eg, DistributedExercise).

ModePanel

The class ApplicationModePanel defines the primary UI element that appears at the bottom of the screen in the software. ApplicationModePanel subclasses must override the getDiagramType method, and this will ensure that the panel appears during the correct mode. The mode panel contains all of the functionality for the mode and diagram that must be handled with a UI. Buttons (usually a check button), and some feedback elements need to be present in the mode panel.

It is important to note that there is always only one mode panel instance of each type, so you must use the activate() method in order to configure the panel appropriately for diagram changes.

In order for a mode panel to be accessible, it needs to be added to the InterfaceConfiguration. This is typically done through the Exercise class.

Actions

DiagramActions are atomic changes that a user can make through the diagram state system. An action takes an existing state and performs a single change to it as a result of something that the user did. Examples of these are using UI elements, entering text into a field, or adding or removing something to a diagram that is being worked on.

Actions are not the only means for making changes to the state. Changes can be made directly using state builders and Diagram.pushState(). However, actions are typically the best way to abstract out changes to diagram state.

Possibilities for future work

As with many projects, this one had many ambitions early on that were not seen through to development, either because of time constraints or because they were out of the scope of the the project. What follows are a list of these possibilities, included in this document to inspire new

directions and future life of the project.

- **Data formatter and editor:** This would allow non-programmers to develop new exercises. The development of a parser and editor is a very significant undertaking, but would be very effective for enabling new types of problems.
- **Define a problem-specification-file specification:** This could be XML, JSON or any other structured format, to describe statics problems in InTEL. This functionality would enable people to create InTEL problems without knowledge of Java programming. A file parser, that converts the problem-specification file to a .jar file containing the problem, would also need to be written. In a more ambitious version of this, allow people to upload models and define the problem using a UI through a web application. The service would then generate a .jar problem exercise file that could be used.