Generic Universal Role Playing System Item Generation System

(GURPS IGS?)

A Project For KSU CIS560

A Collaboration between

Nicholas Sixbury and Sarah Diener

Introduction:

GURPS, or the Generic Universal Role Playing System, is a tabletop roleplaying game that has been published by Steve Jackson Games since 1986. It offers extreme flexibility as a system due to its large number of published material supporting the game along with its very generic and adaptable rule sets. Now, while it's great that the system offers a lot of optional depth, many of the more realistic rule options in particular can become cumbersome to keep track of with pencil and paper.

One fantastic example of GURPS's sometimes cumbersome-to-keep-track-of rules comes from the article "The Deadly Spring" by Douglas H Cole, published in Pyramid 3rd Edition, Volume 33, Low-Tech. In the article, Cole describes an optional system by which players may design their own custom bow and arrows, calculating or customizing draw force, working length of the bow, target draw length, bow components and materials, cross section and shape of bow staff, whether to use a compound or crossbow, among several others and derive game statistics from these options. While the customization and realism can be great, it also involves a great deal of math and planning, such as the formula shown below, taken from the article, used for calculating the maximum draw distance of a particular bow, shown to the right.

$$S = p \times (R + L) - (p - 1) \times [R + [2 \times L \times \sin(\theta/2)]/\theta]$$

Here, R equals length of riser, in inches; R + L is the total length of the bow. The variable p equals number of loops of string, usually three in a compound bow and one in a regular bow. Working string length (S) equals total bow height (R + L), if p = 1. L is the working length of the bow.

θ equals the angle subtended by the chord of the circle made by bending the working parts of the staff, ignoring the riser. It is approximately equal to 8 × δ/L for δ/L less than 10%. Look up δ/L on the *Nasty Transcendental Equation Table*, below, for deflections larger than 10% of the bow's length.

$$D_{max} = \delta + \textit{square root of} \left(\frac{S^2}{4} - \left[\frac{R}{2} + \frac{L \times \sin(\theta/2)}{\theta} \right]^2 \right)$$

For a fully working straight bow, S = L and R = 0.

Nasty Transcendental Equation Table

Look up δ/L on the table below to find the extent to which your bow stave is turning into a circle! As the angle of the bow gets closer to a semicircle (3.14 radians, or 180°), each small increase in δ/L changes θ a great deal. Deflections *greater* than a full semicircle are rare, even for highly reflexed composite bows, though they can be done – the stiff tips (*siyahs*) on some ancient composite bows allow exactly this!

The equation for δ/L > 10% is δ/L = [1 - cos(θ/2)]/θ; this can be solved numerically in a spreadsheet if you don't want to use the table.

Deflection/Length (δ/L)	*Theta (θ)*
0.01	0.08
0.02	0.16

Figure 1: Equation For Calculating Maximum Draw Distance, Pyramid 3-33 pg 7

Now, while some aspects of the game can require a good deal of time calculating things out and planning, a good deal of this can be mitigated using computers. Another example of material for the game that can require some time calculating things out and rolling dice comes in the form of the book "GURPS 4th – Dungeon Fantasy 8 – Treasure Tables". The book allows for random generation of thematic and appropriately stat-ed items for a wide variety of environments and situations, and all it requires is a bit of dice rolling. Well, for better or worse, the book is nearly 60 pages of large tables and rows upon rows of statistical information on various common and uncommon items.

The book starts with a nine-paragraph description of how to actually use all the tables, and right after that is a large table, taking up four full pages, used simply to randomly select a category of item. While the book is carefully cross referenced, it can still be a bit of a chore to flip back and forth several times across the 60+ page volume just to generate a single item. Now, many GURPS books would be horrifically complicated to computerize, but Dungeon Fantasy Treasure Tables, relatively speaking, is much simpler. It simply involves going back and forth between a large amount of organized data while rolling dice for randomness. The data can be stored in a database, in tables similar to those in the book, and the dice can be simulated by computer algorithms as well.

This is where the GURPS Item Generation System comes in. It automates the process of going through the tables by handling all of that processing digitally, and even comes with the added bonus of sharing stat tables created by the community to other players, similarly to sites like GURPS-Repository or the official GURPS forums, which already allow users from all over the world to share materials for the game. With the GURPS Item Generation System, users can both make use of a great system to generate random items for their game, or share custom-made tables with other players.

Technical Description:

The GURPS Item Generation System, or GURPSIGS, is a combination of C# code, SQL databases, and React JavaScript webpages. React is a JavaScript library used to build websites, so it comprises our frontend. Our website is organized into a folder for each page of the website, splitting most pages into multiple components for better readability. In order to get the data from the backend, our API uses get requests. The web API is written in C# and is comprised of many controllers, one for each request that can be sent from the frontend. Several of the controllers needed custom structs in order to turn the json string of data from the frontend into something the backend could use (for example, creating a NewItem struct and deserializing the json string into that struct to pass it to the method that adds a new item to the database).

In terms of development environments, Microsoft Visual Studio was used for the C# and Web-API aspects, Microsoft SQL Server Management Studio 18 was used to interact with the systems database, and Microsoft Visual Studio Code was used for designing webpages. A variety of Microsoft NuGet packages are used in the C# code, along with a package for JSON functionality in order to facilitate communication between the Web-API and client browsers. Additionally, a project titled “DataAccess” was provided by John Keller, the instructor of the course, in order to facilitate database access from c#. It should be noted however, that while much of the tools used to develop the application are Microsoft based, the client webpage itself should run in most standard web browsers on common operating systems.

In terms of the overall system design, we adopted an “MVC-ish” approach, separating application logic, database access, and frontend views from each other. Several different structures of similar classes were also used throughout.

So called “Model” classes served as temporary storage containers for records from the database, allowing information to be passed around as objects instead of a mess of various similarly named properties for every table in the database.

SQL code was also separated out into different categories and folders, with *Setup* scripts being intended to only be run once in order to set up the database, *Create* scripts being used to add new rows to the database, *Delete* scripts being used to delete certain reference entries in the database, *Select* scripts being used to retrieve many rows at a time from a particular table in the database, and *Save* scripts being intended to update information already in the database. There were also four *Report* scripts which used the advantages of database access to generate various report information about the data in the database.

Database Design:

Our database is split into two different schemas in order to reflect the difference in intended use case. The AppRecords schema holds information that is meant to be added and kept indefinitely, being read from more often than its written to. It holds all user information, along with all information on generating items, enchantments, and embellishments. The GeneratedItems schema holds information on items that have already been generated. Tables located in the GeneratedItems table are thought of as in a particular user's inventory. This is an important distinction, as conceptually, the Item, Enchantment, and Embellishment tables in the AppRecords schema are linked more to a particular category, each of which, while linked to a particular user, is more or less accessible to most users.

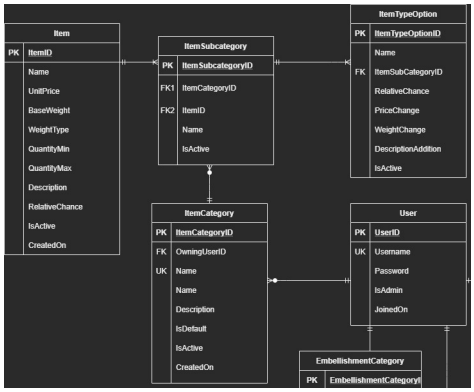

Figure 3: Tables in the AppRecords schema

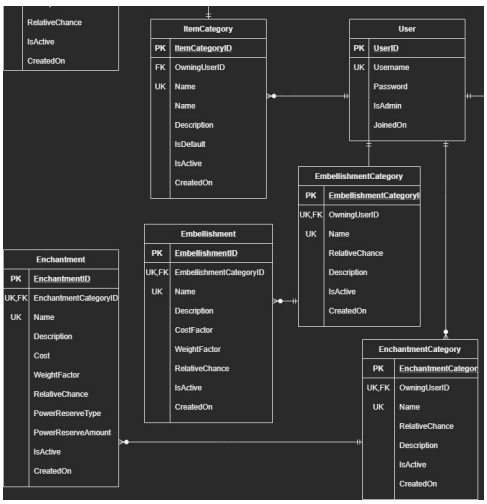

Figure 2: Tables in the AppRecords schema

The inventory tables, however, such as InventoryItem, CreatedEnchantment, and CreatedEmbellishment, are entries that are more free-floating, as CreatedEnchantment and CreatedEmbellishment tables aren't linked to any category or user, instead linked by reference tables EnchantmentRef and EmbellishmentRef. InventoryItems in particular have most of the stats of the Item table, but loses all link to the original item or ItemCategory table. Instead, each item has a link to the user who generated it. While the CreatedEnchantment and CreatedEmbellishment tables are usually just read from, the InventoryItem, EnchantmentRef, and EmbellishmentRef tables are all intended to be written to more often than being read from, relatively speaking.

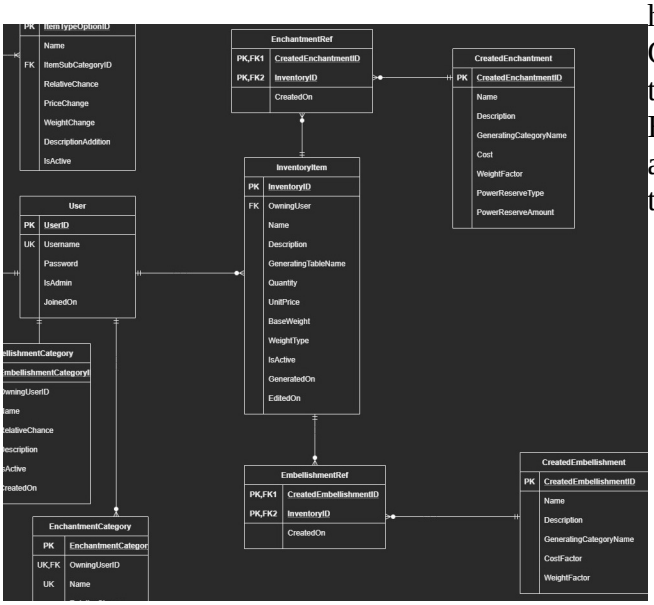

Figure 4: InventoryItem, EnchantmentRef, EmbellishmentRef, CreatedEnchantment, CreatedEmbellishment tables in GeneratedItems schema

System Design:

Within the backend, *DataDelegate*, *Repository*, and *Controller* classes make up a large part of how the system makes it relatively easy for the Web-API to access or perform operations on the database. *Repository* classes are used in order to quickly and painlessly perform set operations on the database, stored as various classes with methods that would each correspond to a particular operation, and would only be concerned with the high level input and output of each procedure. Each *Repository* class corresponds to a table in the database, with each method in that class corresponding to a procedure written in SQL. The *Controller* classes serve in a somewhat similar role to *Repositories*, but instead of being used by the backend as a front-facing data-access tool, the *Controller* classes are used to interface between client web browsers and the backend data, with each one using a *Repository* object in order to do its job.

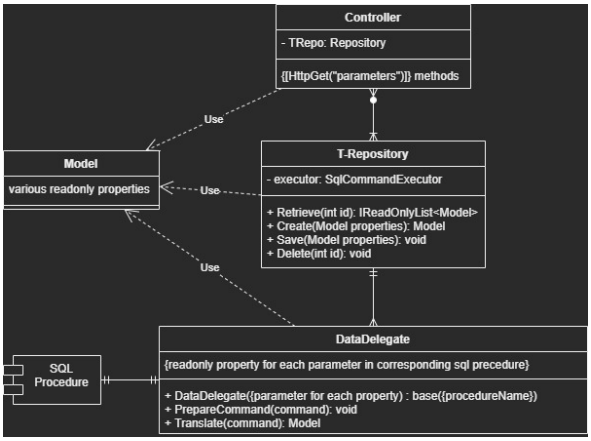

Figure 5: A hybrid diagram of how the core class types relate to each other.

DataDelegates, on the other hand, act as a connecting piece between the front-facing *Repositories*, raw SQL procedures, and various classes in the DataAccess project, which facilitates and largely handles actual command execution. *DataDelegates* in particular make up a large part of the code base, as in order for the frontend (or even most of the backend) to link up with the database for a particular operation, a *DataDelegate* class must be written (with our current system) for each and every SQL procedure in the database. This means that to fully implement every part of every operation, a *DataDelegate* class must be written for each of our systems *eighty three* SQL procedures, defining both the parameters and return data for each and every single procedure.

System Features and Usage:

When a user first visits the site, they're greeted by a fairly minimal page showing account login or creation buttons, and a blank table titled "Original GURPS Generator". Without logging in, a user can use this page to generate a set number of randomly generated items, selected from all the items added by the administrator as default, taking the relative chance of each item into account. Clicking the "Create Account" button lets a user create a username and password which can then be used to

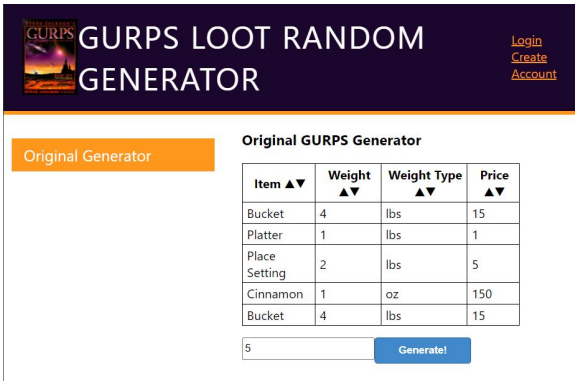

Figure 6: Item generation screen if not logged in

login as that account in the future. The Login button, of course, logs the user into the account with matching username and password.

The accounts are important in particular because user-created tables and items are tied to user accounts, and inventory is also tied to user accounts, so an account allows a user to create their own content, view the content of others, and save an inventory of the items they've generated. Various lifetime statistics are also saved for each user, so that's an advantage as well. Users also need their own account in order to create their own tables and items, which can then be used by other users.

Figure 7: Login screen

The system has both great strengths and great weaknesses. In terms of strengths, the system we've created for managing the database and defining new behavior is highly extensible and segmented, making it relatively simple for a system of its size to add/remove/modify features. It also bundles a high degree of functionality into a fairly simple package, as the site is very easy to use, even for a new user who hasn't seen it before.

Figure 8: Example statistics information

The system has great weaknesses as well though. Aside from the fact that we were unable to include all of the functionality we intended (not really handling enchantments or embellishments, for example), there is little to no error handling, and the user account security is frankly abysmal. Even though the application explicitly allows for users to create their own objects with strings that are stored in the database, that input is not sanitized at all, meaning that SQL injection is probably not particularly difficult. In addition, user login name is displayed publicly, and passwords are not sent and stored in plan text. Even the database connection isn't encrypted, meaning that passwords could be easily leaked.

However, despite this, one strength of the system is that the weaknesses are things that could be changed without a major redesign. They're areas for future improvement, and because of the strong versatility of the system, additional functionality to check for errors or encrypt passwords isn't too complicated.

Report Queries:

```
CREATE OR ALTER PROCEDURE AppRecords.ItemCategorySummary
AS
SELECT IC.ItemCategoryID, IC.[Name], IC.[Description],U.Username AS OwningUser,
COALESCE(AVG(I.UnitPrice), 0) AS AverageCost, COALESCE(AVG(I.BaseWeight), 0) AS AverageWeight,
COALESCE(SUM(I.UnitPrice), 0) AS TotalCost, COALESCE(SUM(I.BaseWeight), 0) AS TotalWeight
FROM AppRecords.ItemCategory IC LEFT JOIN AppRecords.ItemSubcategory ISC ON
IC.ItemCategoryID = ISC.ItemCategoryID LEFT JOIN AppRecords.Item I ON
I.ItemID = ISC.ItemID LEFT JOIN AppRecords.[User] U ON
U.UserID = IC.OwningUserID
GROUP BY IC.ItemCategoryID, IC.[Name], IC.[Description], IC.OwningUserID,
    IC.CreatedOn, U.Username;
GO
```

Item Category Summary

Statistics information for every ItemCategory, including the creating user, number of items in the table, average cost and weight of items in the table, total cost and weight of all items in the table, createdOn date, sorted by least recent creation.

Results | Messages

	ItemCategoryID	Name	Description	OwningUser	AverageCost	AverageWeight	TotalCost	TotalWeight
1	1	Spices	A collection of different spices.	Administrator	112	1	3717	33
2	2	Cooking	A collection of items useful for cooking.	Administrator	22	3	508	90
3	5	An interesting table name	Description	Administrator	0	0	0	0
4	3	Yeets	Description	sdiener	0	0	0	0
5	4	Flamingos	Description	sdiener	850	2	1700	5

```
CREATE OR ALTER PROCEDURE AppRecords.ItemEnhancementSummary
    @ItemCategoryID INT,
    @EnchantmentCategoryID INT,
    @EmbellishmentCategoryID INT
AS
SELECT I.ItemID, I.[Name] AS ItemName, EN.EnchantmentID, EN.[Name] AS EnchantmentName,
EM.EmbellishmentID, EM.[Name] AS EmbellishmentName,
((I.UnitPrice * (1 + EM.CostFactor)) + EN.Cost) AS Cost,
(I.BaseWeight * (1 + EM.WeightFactor + EN.WeightFactor)) AS [Weight]
FROM AppRecords.Item I INNER JOIN AppRecords.ItemSubcategory ISC ON
I.ItemID = ISC.ItemID INNER JOIN AppRecords.ItemCategory IC ON
ISC.ItemSubcategoryID = IC.ItemCategoryID INNER JOIN AppRecords.Enchantment EN ON
EN.EnchantmentCategoryID = @EnchantmentCategoryID INNER JOIN AppRecords.Embellishment EM
ON EM.EmbellishmentCategoryID = @EmbellishmentCategoryID
WHERE IC.ItemCategoryID = @ItemCategoryID;
GO
```

Item Enhancement Summary

For all items in a particular table, show each combination of variations of single embellishments and enchantments from selected embellishment and enchantment categories, including a final cost and weight for each item, also including category of each item, embellishment, and enchantment in the row they belong in.

	ItemID	ItemName	EnchantmentID	EnchantmentName	EmbellishmentID	EmbellishmentName	Cost	Weight
1	1	Allspice	1	Frost	1	Double	350.00	1.00
2	1	Allspice	1	Frost	2	Triple	500.00	1.00
3	1	Allspice	2	Fire	1	Double	360.00	1.00
4	1	Allspice	2	Fire	2	Triple	510.00	1.00

User Inventory Summary

List all items that have been generated for a particular user, including information on most and least recent generation date, cost and weight of each item, the category each item is from, the name of the item, and number of users which have generated them. Items are ordered descending by the number of users who have that item in their inventory.

```
CREATE OR ALTER PROCEDURE GeneratedItems.UserInventorySummary
    @UserID INT
AS

SELECT II.[Name], II.GeneratingTableName, MIN(II.GeneratedOn) AS EarliestGeneration,
MAX(II.GeneratedOn) AS LatestGeneration, II.UnitPrice, II.BaseWeight,
(COUNT(II.InventoryID) * II.Quantity) AS NumberGenerated
FROM GeneratedItems.InventoryItem II /*LEFT JOIN GeneratedItems.EmbellishmentRef EMR
ON II.InventoryID = EMR.InventoryID LEFT JOIN GeneratedItems.EnchantmentRef ENR
ON II.InventoryID = ENR.InventoryID LEFT JOIN GeneratedItems.CreatedEmbellishment CEM
ON EMR.CreatedEmbellishmentID = CEM.CreatedEmbellishmentID LEFT JOIN
GeneratedItems.CreatedEnchantment ON ENR.CreatedEnchantmentID = ENR.CreatedEnchantmentID*/
WHERE II.OwningUserID = @UserID
GROUP BY II.[Name], II.GeneratingTableName, II.UnitPrice, II.BaseWeight, II.Quantity;

GO
```

```
1  --just display inventory using report query
2  EXEC GeneratedItems.UserInventorySummary @UserID = 3;
3
```

100 %

Results | Messages

	Name	GeneratingTableName	EarliestGeneration	LatestGeneration	UnitPrice	BaseWeight	NumberGenerated
1		MissingCategoryName	2021-11-29 22:00:18.6778508 +00:00	2021-11-29 22:07:58.8863982 +00:00	900	3	6
2		MissingCategoryName	2021-11-29 21:56:50.2198986 +00:00	2021-11-29 21:56:50.2198986 +00:00	900	3	4
3		MissingCategoryName	2021-11-29 22:05:06.2977864 +00:00	2021-11-29 22:12:18.6274183 +00:00	900	3	20
4	Anise	MissingCategoryName	2021-11-29 11:01:16.4830056 +00:00	2021-11-29 22:03:53.4051681 +00:00	150	1	4
5	Annatto	MissingCategoryName	2021-11-29 22:03:53.9084460 +00:00	2021-11-29 22:03:53.9084460 +00:00	113	1	3
6	Asafetida	MissingCategoryName	2021-11-29 11:01:10.8242471 +00:00	2021-11-29 11:01:10.8242471 +00:00	75	1	1
7	Asafetida	MissingCategoryName	2021-11-29 22:03:53.7826005 +00:00	2021-11-29 22:03:53.7826005 +00:00	75	1	2
8	Clove	MissingCategoryName	2021-11-29 11:01:16.2939422 +00:00	2021-11-29 11:01:16.2939422 +00:00	150	1	3
9	Coriander	MissingCategoryName	2021-11-29 11:01:16.7180450 +00:00	2021-11-29 11:01:16.7180450 +00:00	150	1	1
10	Cumin	MissingCategoryName	2021-11-29 17:55:48.4676673 +00:00	2021-11-29 17:55:48.4676673 +00:00	150	1	3

User Item Summary

For each user, show number of tables they've created, number of items they've created, number of categories generated from, number of items generated, and the day they joined.

```
CREATE OR ALTER PROCEDURE AppRecords.UserItemSummary
AS
SELECT U.UserID, U.Username, COUNT(DISTINCT IC.ItemCategoryID) AS TablesCreated,
COUNT(DISTINCT I.ItemID) AS ItemsCreated, COUNT(DISTINCT II.GeneratingTableName) AS TablesUsed,
COUNT(II.InventoryID) AS ItemsGenerated, U.JoinedOn
FROM AppRecords.[User] U LEFT JOIN AppRecords.ItemCategory IC ON
U.UserID = IC.OwningUserID LEFT JOIN AppRecords.ItemSubcategory ISC ON
ISC.ItemCategoryID = IC.ItemCategoryID LEFT JOIN AppRecords.Item I ON
ISC.ItemID = I.ItemID LEFT JOIN GeneratedItems.InventoryItem II ON
II.OwningUserID = U.UserID
GROUP BY U.UserID, U.Username, U.JoinedOn
GO
```

Results | Messages

	UserID	Username	TablesCreated	ItemsCreated	TablesUsed	ItemsGenerated	JoinedOn
1	3	sdiener	2	2	1	153	2021-11-29 11:00:32.2866172 +00:00
2	1	Administrator	3	56	1	1311	2021-11-29 10:45:34.4782352 +00:00
3	4	username	0	0	0	0	2021-11-29 11:47:38.4562368 +00:00
4	5	whatev	0	0	0	0	2021-11-29 21:20:34.3550349 +00:00
5	2	A Deleted User	0	0	1	77	2021-11-29 10:45:34.4824383 +00:00

Summary and Discussion:

Obviously, we were not able to implement the entirety of our original vision for the project. Several QoL features didn't make it into our current version, and we basically gave up on enchantments and embellishments just because the ambitious scope of our project went beyond our time constraints. In fact, we had hardly anything in the way of polished available features when we made our presentation of the application on 11/29/21. However, now that we've had a little bit more time to connect things, we have almost all of the functionality for items, and the finished product is something extremely usable.

We've both learned a bunch about implementing Web-APIs and database access in C#, simply due to the time we've spent working on them. We've also learned a bit about project scope, as being too ambitious for the time frame we had available likely cost us time that we could have used to make a more polished product. In terms of what we might change, it's hard to say whether it would be better to reduce scope or just start working earlier. Considering the reasons we both didn't get manage to start as soon as we might've liked to, reducing project scope is probably better.

There's a lot of room for future improvements and functionality. As said previously, we've created a great system with clear structure, and enough modularity to easily add new features or make improvements to what's already there. In particular, it would be good to use Microsoft's database tools to encrypt our connections for better security. Error handling would also be a good idea. In addition to that, it would be great if we could add smoother editing of information on the user's end, and make sure that users generally have better ways to manipulate the data they input onto the site.

END

By Nicholas Sixbury and Sarah Diener