The Budget Buddy

Design Phase

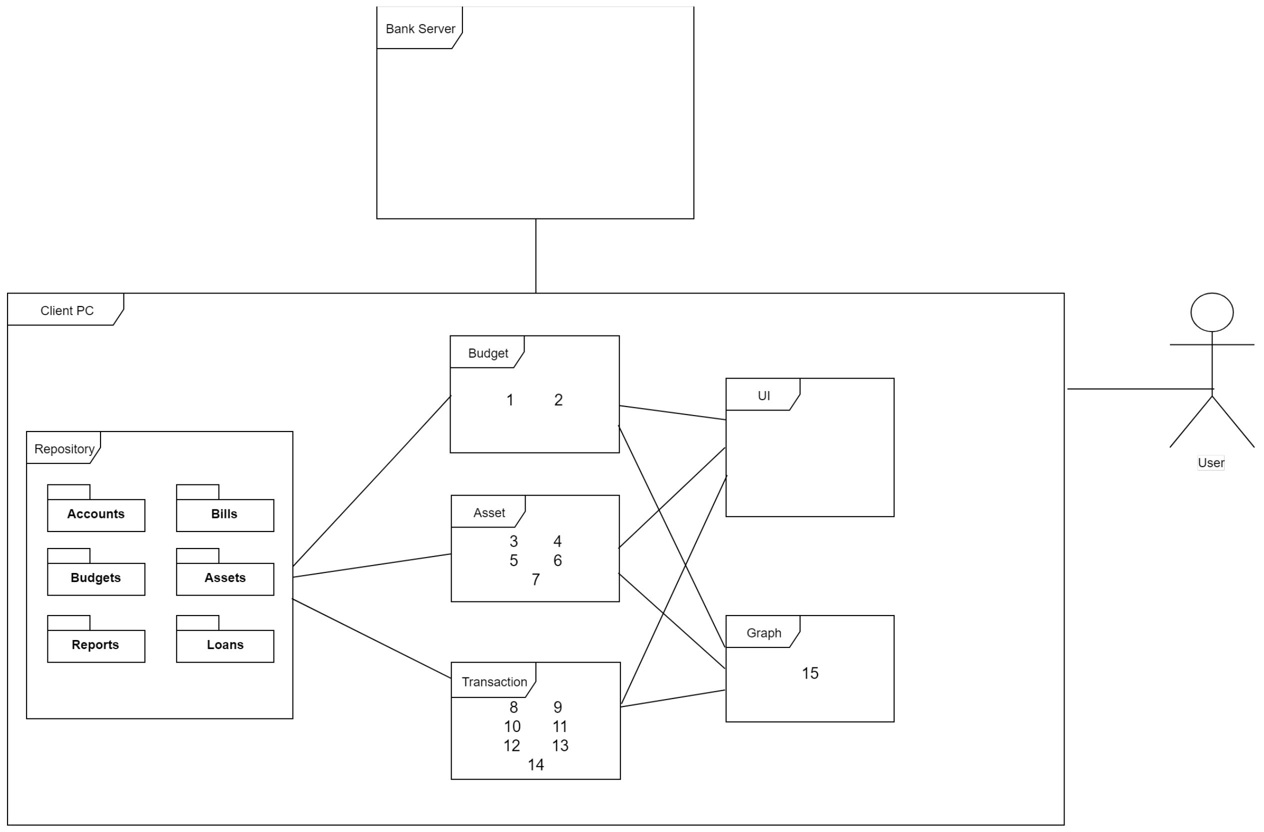
Team Name: The Accountants

James Hall - Team Leader
(jgh0020@auburn.edu)
Ryan McWilliams - Requirements Engineer
(rsm0015@auburn.edu)
Jack Scott - Architect
(jds0076@auburn.edu)
Chris Wheeler - Designer
(cgw0014@auburn.edu)
Taylor Brown - Programmer
(tcb0027@auburn.edu)
Brodderick Rodriguez - Programmer
(bcr0012@auburn,edu)

Table of Contents

Architectural Design

UML Deployment Diagram



Budget	Asset	Transaction	Graph
1. Create budgets 2. Create budget reports	3. Record assets 4. Display current value of assets 5. Loan tracking (amortization calendar) 6. Calculate net worth 7. Tracking savings and net worth over time	8. Create financial reports 9. Export financial reports 10. Record transactions 11. Update transaction records 12. Categorization of transaction history 13. Automatic bill pay 14. Bill pay reminder	15. Create graphs

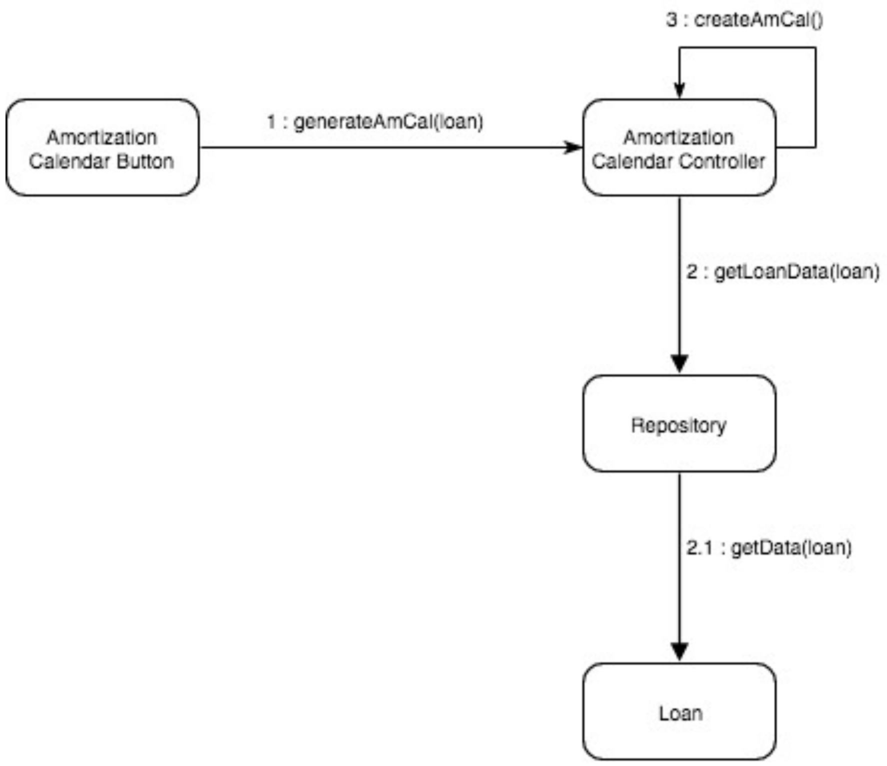
Architectural Design Rationale

We decided to use the repository architecture because we have a lot of shared data in our system. Almost all of the data in our system needs to be accessible by any other part of the system. Each individual subsystem doesn’t do a lot of processing on the data, so it can be easily stored in a widely understood format. The main kind of processing we do on the data is searching through it to see if it is relevant to the current needs of the user and then formatting it in a way that the user can easily understand. The main downside to using this format is that the repository is a very highly coupled part of the system, so maintainability will be sacrificed if there needs to be a major change to the repository layout.

Detailed Design

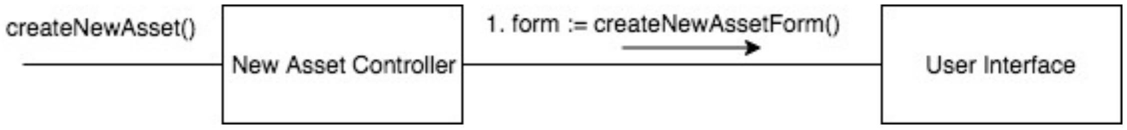
Interaction Design

Create Amortization Calendar



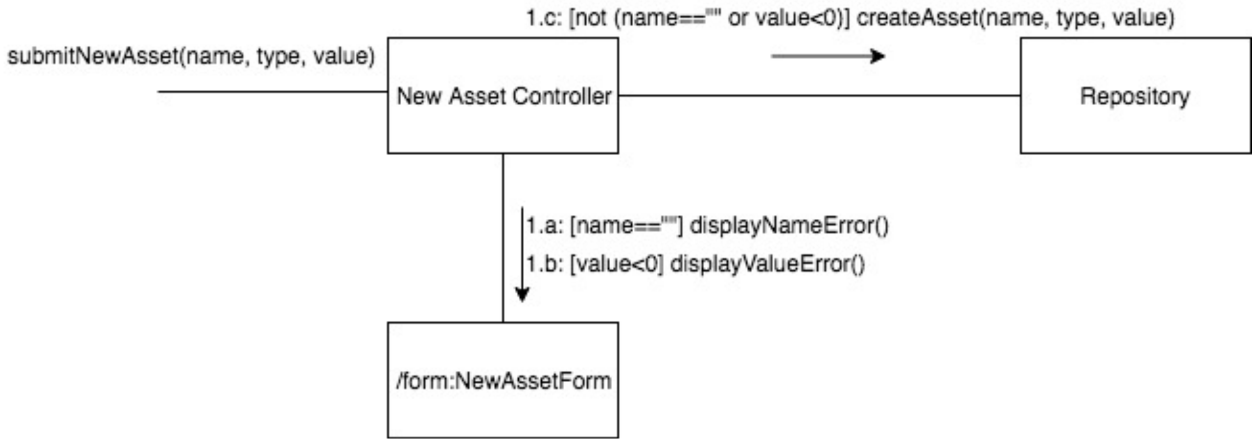
The controller pattern is used by the Amortization Calendar Controller, which is created by the Amortization Calendar Button using the creator pattern. The controller coordinates the operations relating to this use case, and the button creates the controller because it has the necessary information to instantiate it. The repository is created using pure fabrication, since it doesn't actually exist in the domain, but its creation allows a lower coupling between the controller and entity objects.

Create New Asset



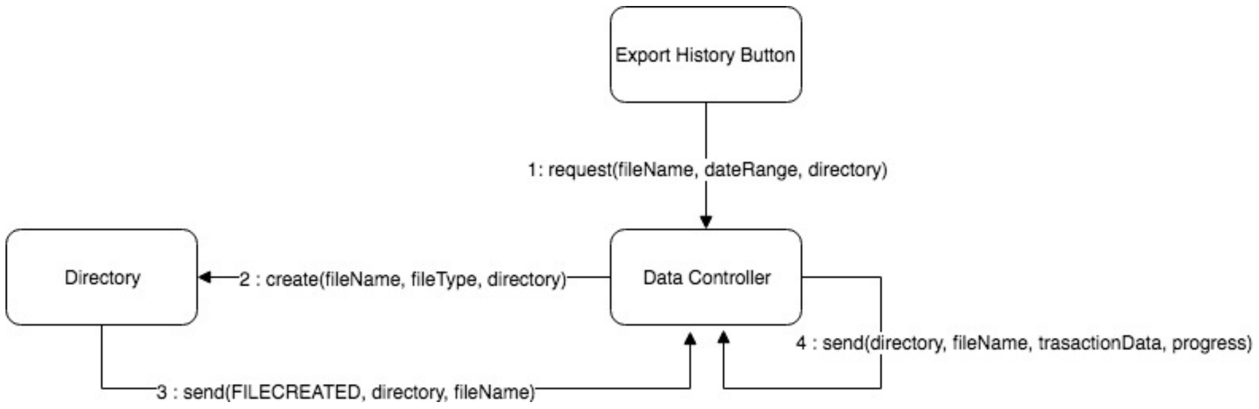
The user interface is a creator because it has the responsibility to create the new asset form. The New Asset Controller is a controller because it is responsible for maintaining the state and data in the create new asset use case.

Submit New Asset


1.c: [not (name=="" or value<0)] createAsset(name, type, value)
submitNewAsset(name, type, value)
New Asset Controller
Repository
1.a: [name==""] displayNameError()
1.b: [value<0] displayValueError()
/form:NewAssetForm

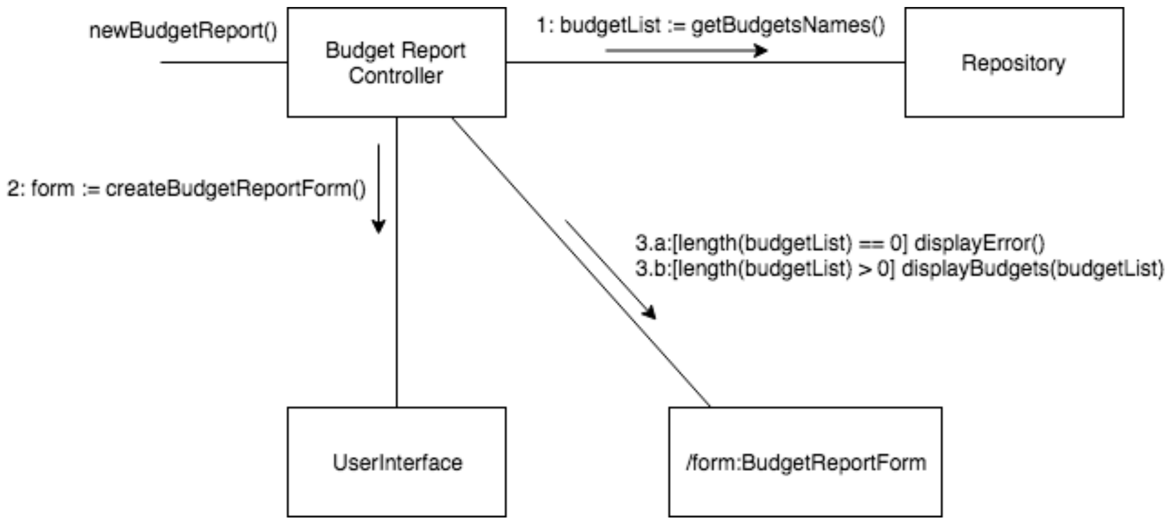
The user interface is a creator because it has the responsibility to create the new asset form. The New Asset Controller is a controller because it is responsible for maintaining the state and data in the create new asset use case.

Gathering Data For File Generation


Export History Button
1: request(fileName, dateRange, directory)
Directory
2 : create(fileName, fileType, directory)
Data Controller
4 : send(directory, fileName, trasactionData, progress)
3 : send(FILECREATED, directory, fileName)

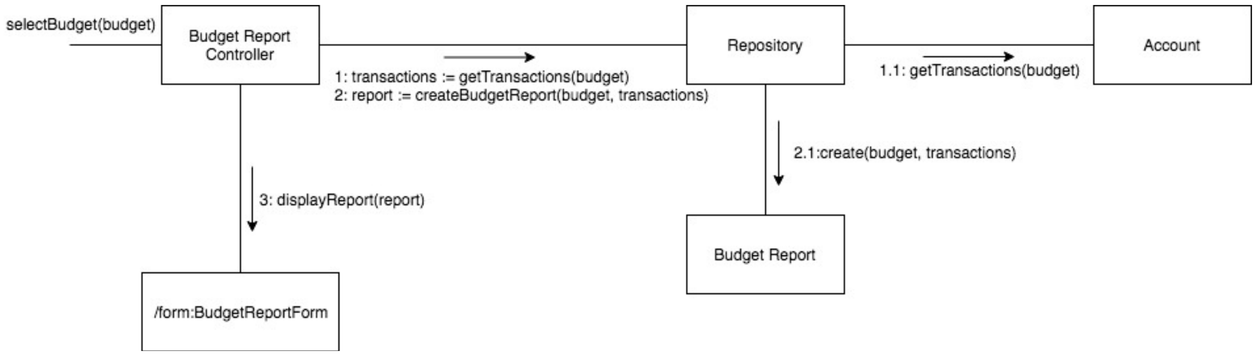
The data controller for the Export Reports use case uses the High Cohesion pattern because the data controller manages virtually all core operations within this use case, with the exception of the “Export Reports” button. The data controller also uses the Controller pattern because it receives a request from the “Export Reports” button UI objects.

New Budget Report


newBudgetReport()
Budget Report Controller
1: budgetList := getBudgetsNames()
Repository
2: form := createBudgetReportForm()
3.a:[length(budgetList) == 0] displayError()
3.b:[length(budgetList) > 0] displayBudgets(budgetList)
UserInterface
/form:BudgetReportForm

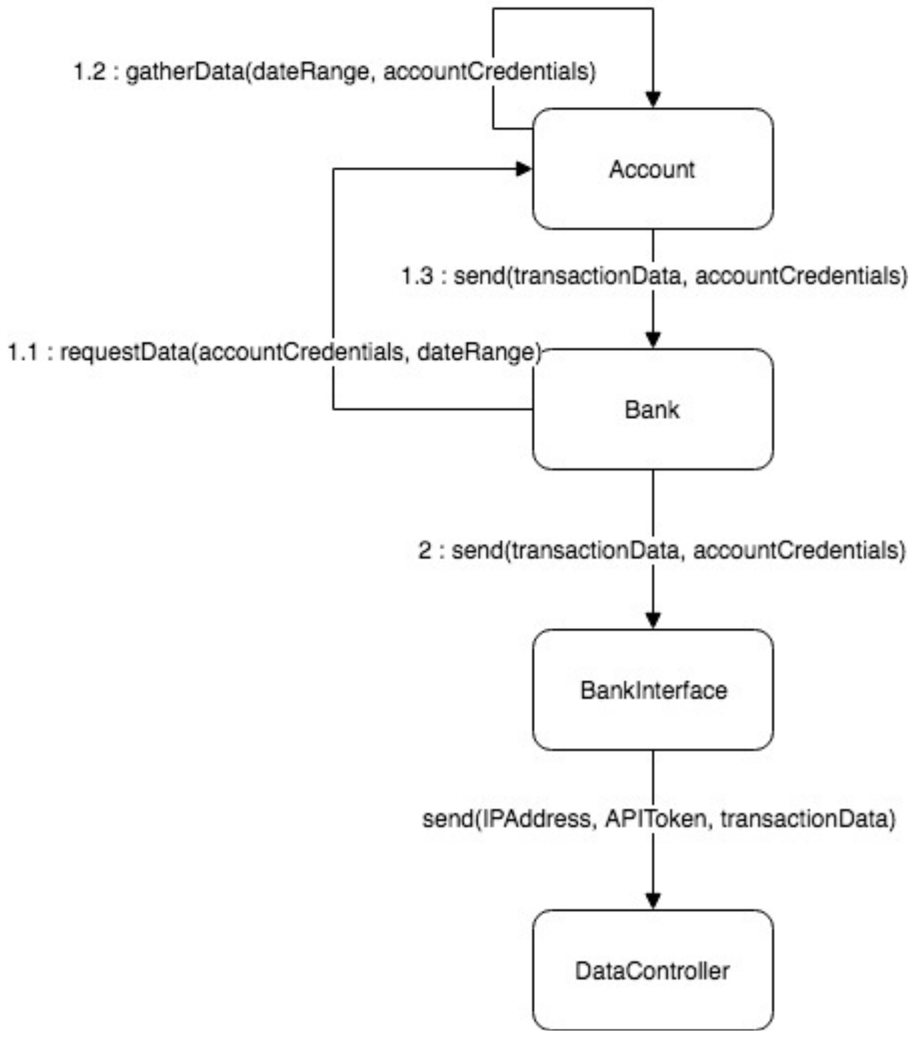
The user interface is a creator because it has the responsibility to create the budget report form. Budget Report Controller is a controller because it is responsible for maintaining the state and data in the create new budget report use case. The repository uses indirection to get data from its budget objects.

Select Budget


selectBudget(budget)
Budget Report Controller
1: transactions := getTransactions(budget)
2: report := createBudgetReport(budget, transactions)
Repository
1.1: getTransactions(budget)
Account
2.1:create(budget, transactions)
3: displayReport(report)
Budget Report
/form:BudgetReportForm

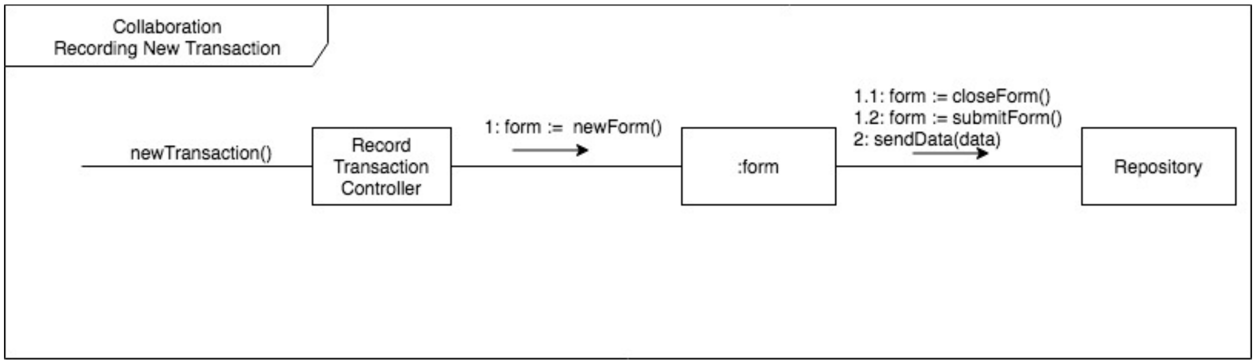
Budget Report Controller is a controller because it is responsible for maintaining the state and data in the create new budget report use case. The repository is a creator because it has the responsibility of creating the budget report class that stores the raw report data. The repository uses indirection to get data from its account objects.

Receiving New Data


1.2 : gatherData(dateRange, accountCredentials)
Account
1.3 : send(transactionData, accountCredentials)
1.1 : requestData(accountCredentials, dateRange)
Bank
2 : send(transactionData, accountCredentials)
BankInterface
send(IPAddress, APIToken, transactionData)
DataController

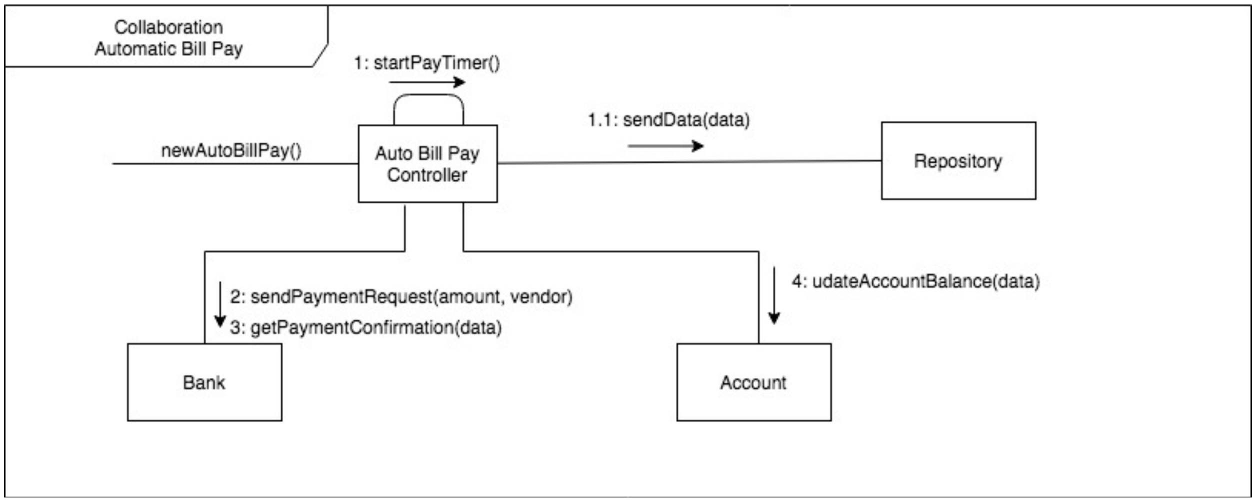
The data controller for the Update Transaction Records use case uses the controller pattern because it receives a request to update transaction records when a user presses the “Open Transaction History” button. The bank data interface entity uses the Pure Fabrication pattern to simplify the duties of the data controller by communicating with the bank.

Record Transaction



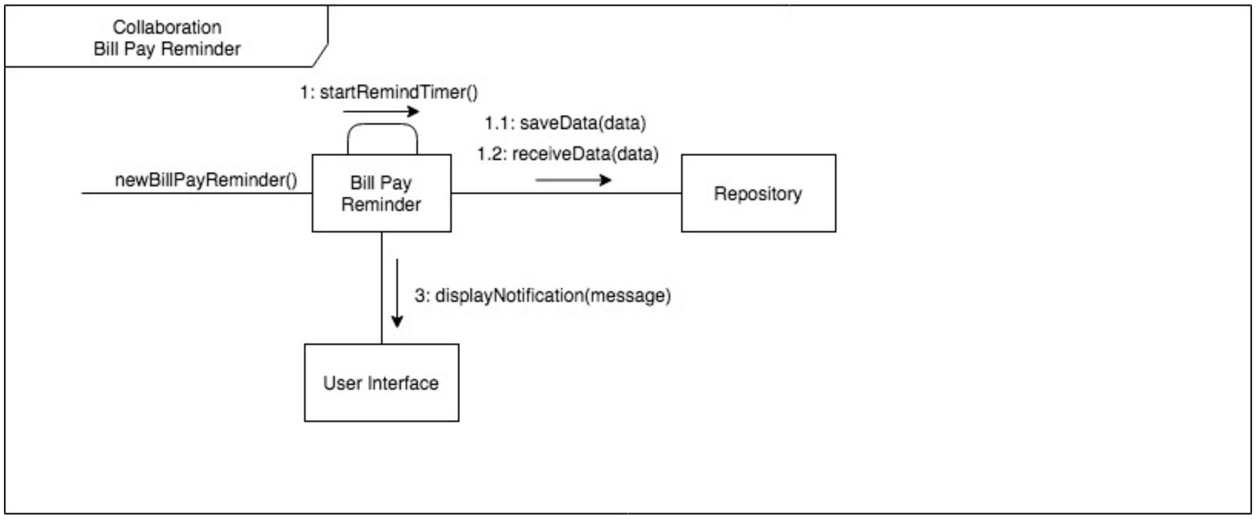
The Record Transaction Controller uses the the controller pattern, which is created by the Record Transaction button that also uses the creator pattern. The Record Transaction Button Controller creates the Record Transaction Controller. The information created is stored in the repository using pure fabrication allowing for low coupling between both controllers.

Automatic Bill Pay



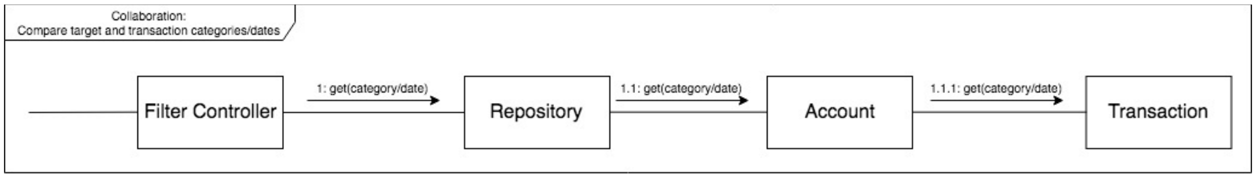
Like the Bill Pay Reminder use case, the Automatic Bill Pay Controller uses the the controller pattern, which is created by the Automatic Bill Pay Button that also uses the creator pattern. The Automatic Bill Pay Button Controller creates the Automatic Bill Pay Controller. The information created by the user is stored in the repository using pure fabrication and checked at the start of the program allowing for low coupling between both controllers.

Bill Pay Reminder



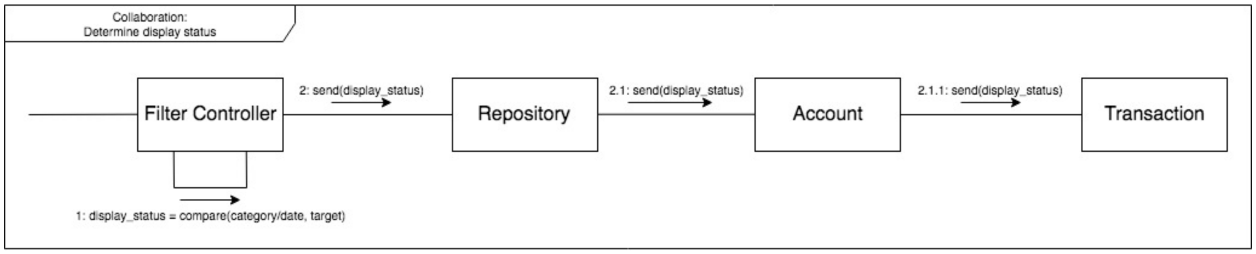
The Bill Pay Reminder Controller uses the the controller pattern, which is created by the Bill Pay Reminder Button that also uses the creator pattern. The Bill Pay Reminder Button Controller creates the Bill Pay Reminder Controller. The information created by the user is stored in the repository using pure fabrication allowing for low coupling between both controllers.

Compare Target and Transaction Categories/dates



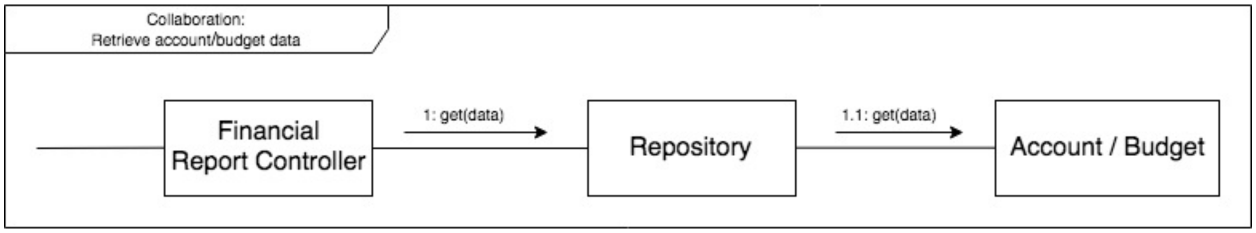
This diagram also demonstrates the Controller pattern for the same reason. It also demonstrates the Pure Fabrication pattern because the Repository class to improve cohesion and coupling between the Filter Controller and the Account and Transaction classes.

Determine Display Status



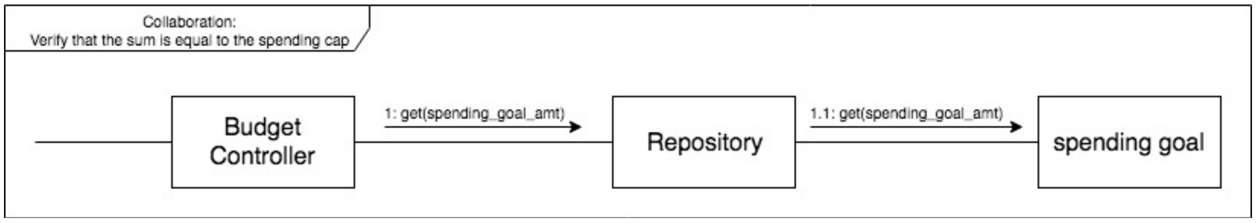
This diagram also demonstrates the Controller pattern. It also demonstrates the Expert pattern because the Filter Controller which is responsible for initiating the event is also the one with the information necessary to then fulfill the responsibility.

Retrieve Account/Budget Data



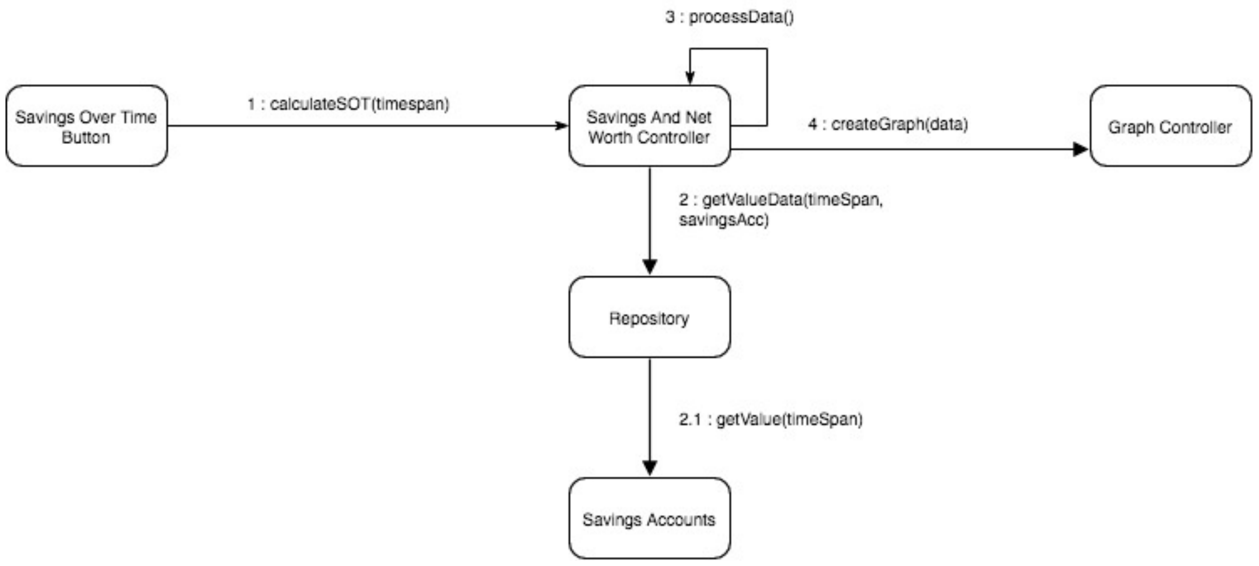
This diagram also demonstrates the Controller pattern. It also demonstrates the Polymorphism pattern because the Account and Budget classes are responsible for defining and retrieving the data appropriate for the Financial Report Controller.

Verify sum is equal to spending cap


Collaboration:
Verify that the sum is equal to the spending cap
Budget Controller
1: get(spending_goal_amt)
Repository
1.1: get(spending_goal_amt)
spending goal

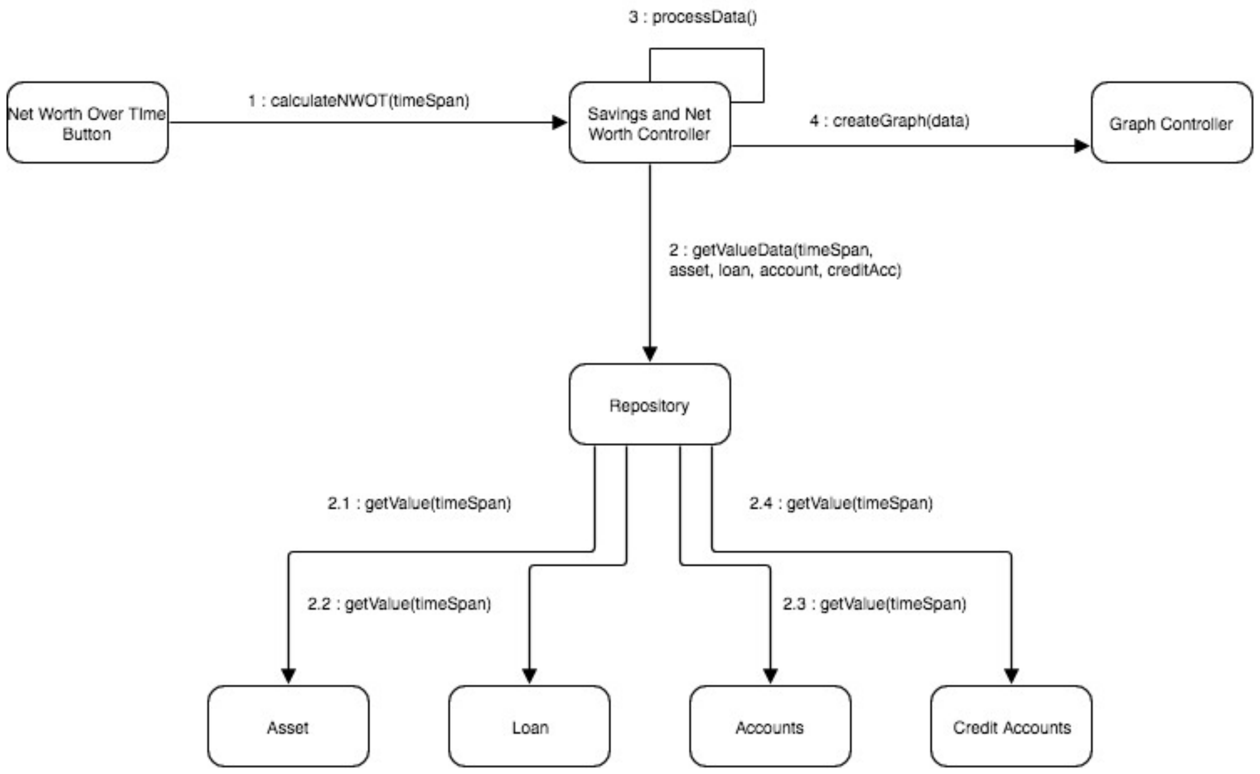
This diagram demonstrates the Controller pattern because the budget controller object is represented as sending the initial signal and receiving what is returned. It also demonstrates the Indirection pattern because the repository functions as a mediator between the Budget Controller and the Spending Goal.

View Savings Over Time



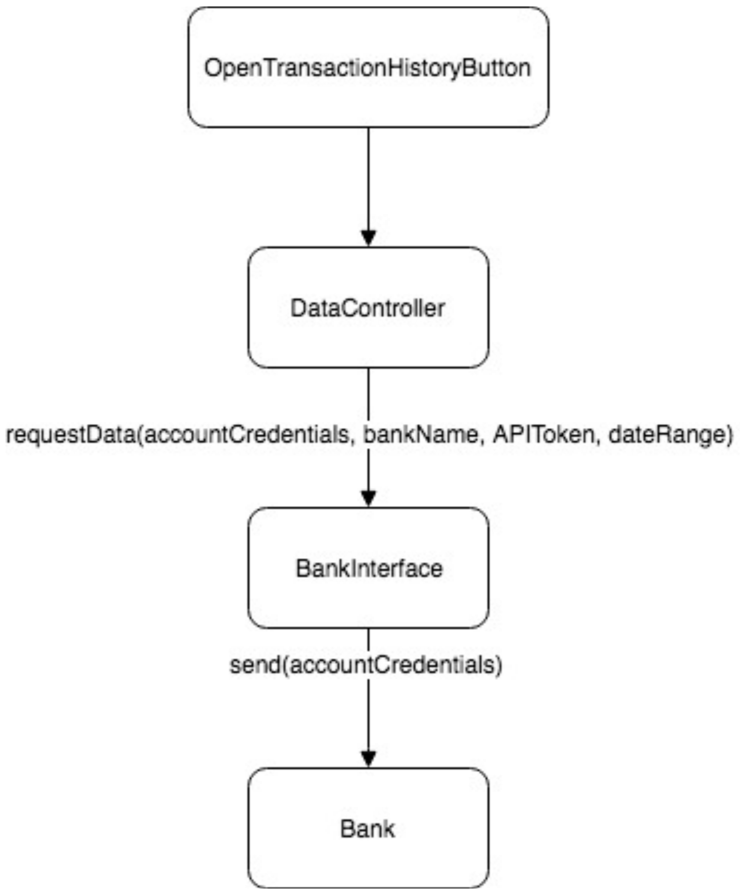
This use case is nearly identical to the previous one and even shares a controller. Therefore, they follow many of the same GRASP patterns. The controller pattern is used by the Savings and Net Worth Controller, which is created by the Savings Over Time Button using the creator pattern. The controller coordinates the operations relating to this use case, and the button creates the controller because it has the necessary information to instantiate it. The repository is an example of pure fabrication as it doesn't actually exist as a concept in the domain, but creating it allows a barrier between the controller and all of the entities. This follows the indirection pattern but also helps to maintain low coupling.

Calculate Net Worth Over Time



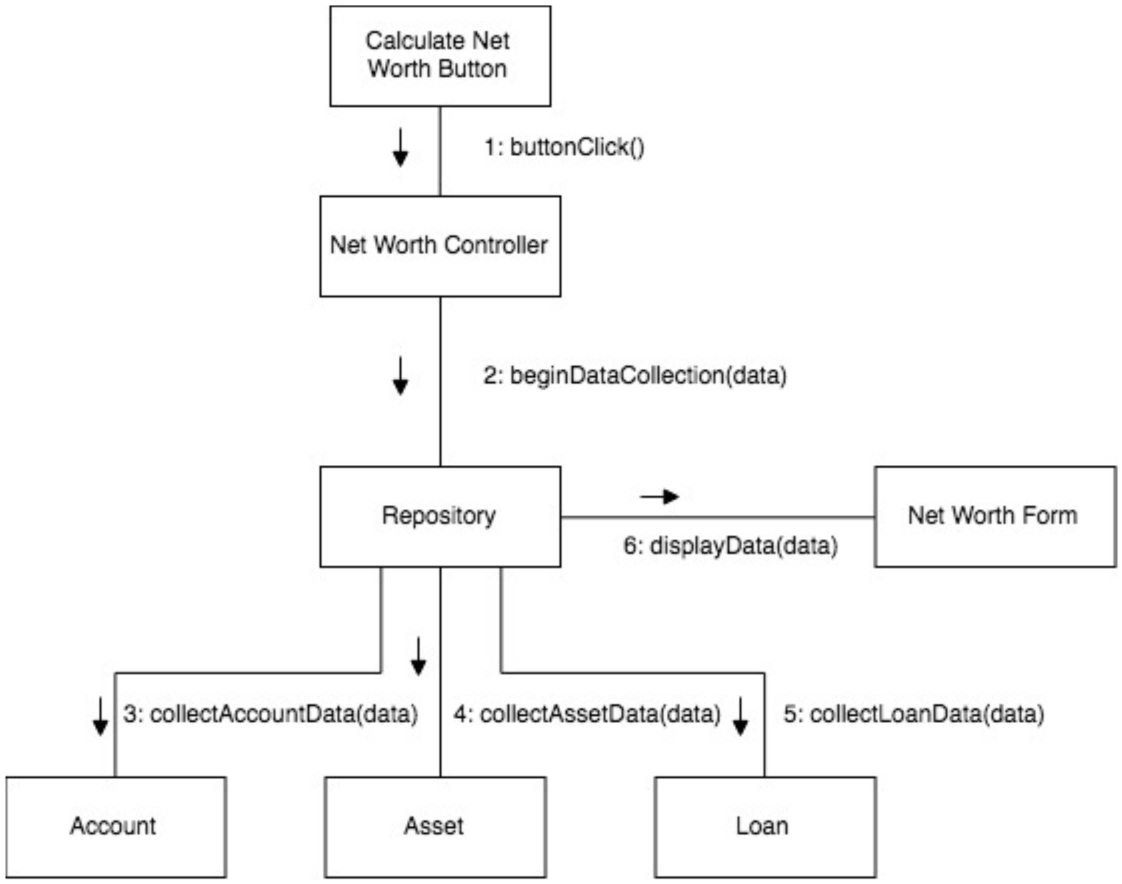
The controller pattern is used by the Savings and Net Worth Controller, which is created by the Net Worth Over Time Button using the creator pattern. The controller coordinates the operations relating to this use case, and the button creates the controller because it has the necessary information to instantiate it. The repository is an example of pure fabrication as it doesn't actually exist as a concept in the domain, but creating it allows a barrier between the controller and all of the entities. This follows the indirection pattern but also helps to maintain low coupling.

Wait For New Data


OpenTransactionHistoryButton
DataController
requestData(accountCredentials, bankName, APIToken, dateRange)
BankInterface
send(accountCredentials)
Bank

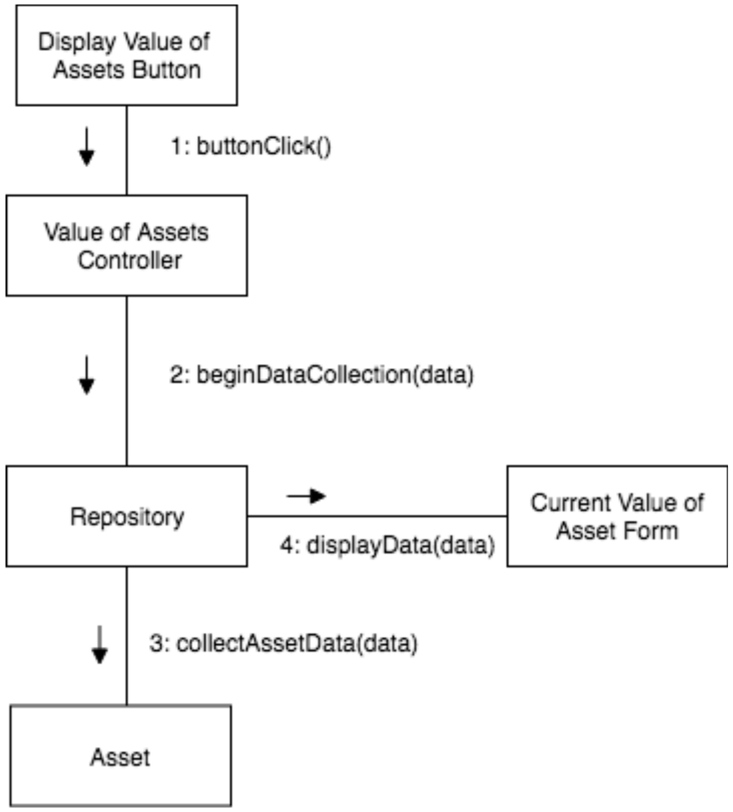
The bank entity in the Update Transaction Records use case uses the Pure Fabrication pattern because the Bank entity simplifies the duties of the Bank Interface entity by communicating directly with the bank accounts. The Account entity uses the Expert pattern because it has complete knowledge of all data contained within the bank accounts.

Calculate Net Worth



The use case Calculate Net Worth follows the GRASP guidelines of Controller and High Cohesion. Because Calculate Net Worth has a single purpose, it follows the GRASP guideline of High Cohesion. Furthermore, because Calculate Net Worth uses a controller, it follows the GRASP guideline of Controller.

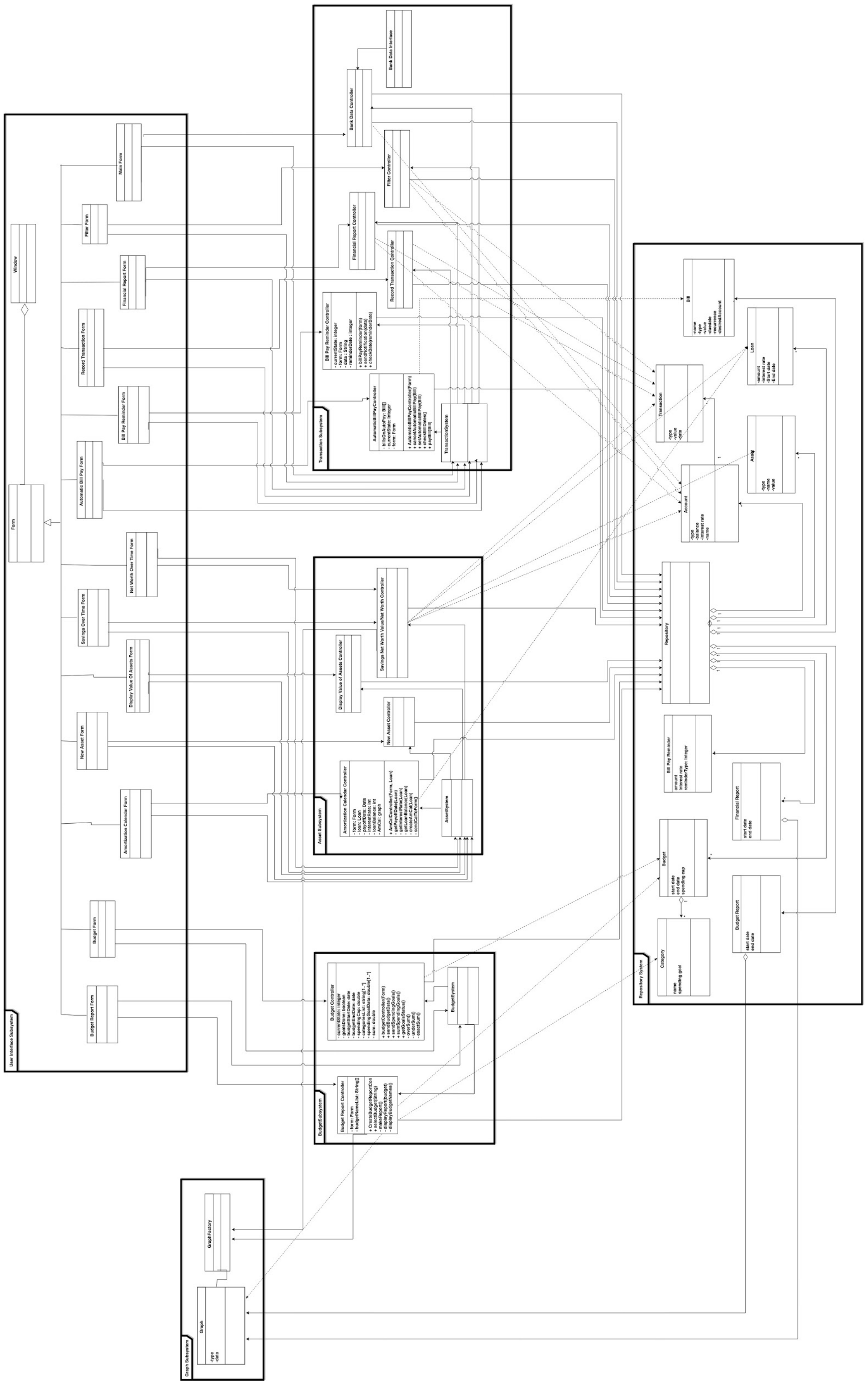
Calculate Current Value of Assets


Display Value of Assets Button
1: buttonClick()
Value of Assets Controller
2: beginDataCollection(data)
Repository
4: displayData(data)
Current Value of Asset Form
3: collectAssetData(data)
Asset

The use case Current Value of Assets follows the GRASP guidelines of Controller and High Cohesion. Current Value of Assets has one specific purpose: to collect all monetary value of a users assets and display it using the Graph class. That process is delegated to a controller. Therefore, Current Value of Assets follows the GRASP guidelines of Controller and High Cohesion.

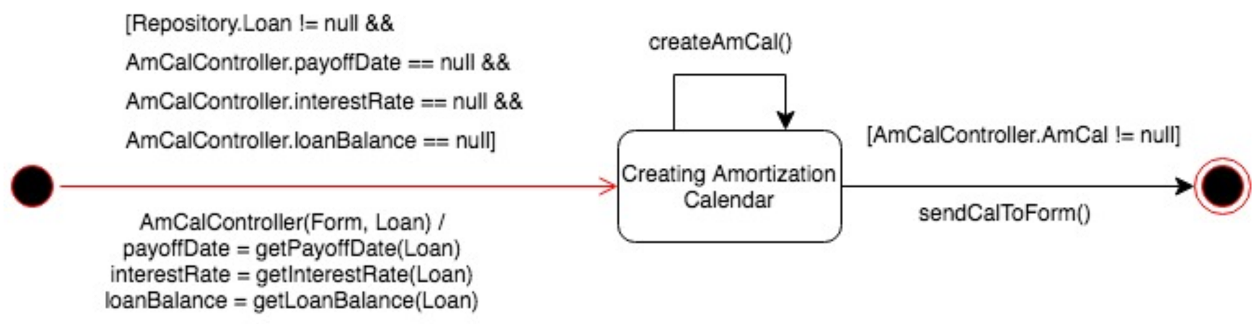
Design Class Diagram

https://drive.google.com/open?id=1MTt_mxYzA5mOQULpRCnC4qhXzy0TTEI1



Class Design

Amortization Calendar Controller


[Repository.Loan != null &&
AmCalController.payoffDate == null &&
AmCalController.interestRate == null &&
AmCalController.loanBalance == null]
AmCalController(Form, Loan) /
payoffDate = getPayoffDate(Loan)
interestRate = getInterestRate(Loan)
loanBalance = getLoanBalance(Loan)
createAmCal()
Creating Amortization Calendar
[AmCalController.AmCal != null]
sendCalToForm()

Class Invariant: this.Form != null && this.Loan != null

Precondition: Repository.Loan != null
+ AmCalController(Form, Loan)
Postcondition: payoffDate != null && interestRate != null && loanBalance != null

Precondition: AmCalController.payoffDate == null
- getPayoffDate(Loan)
Postcondition: AmCalController.payoffDate != null

Precondition: AmCalController.interestRate == null
- getInterestRate(Loan)
Postcondition: AmCalController.interestRate != null

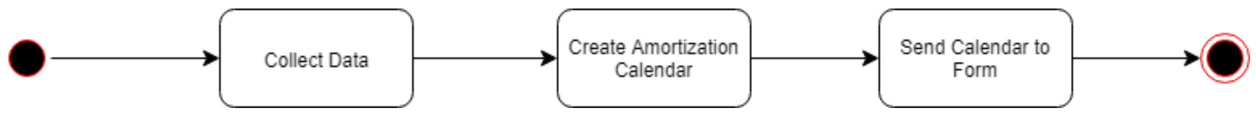
Precondition: AmCalController.loanBalance == null
- getLoanBalance(Loan)
Postcondition: AmCalController.loanBalance != null

Precondition: AmCalController.payoffDate != null &&
AmCalController.interestRate != null &&
AmCalController.loanBalance != null
- createAmCal(Loan)
Postcondition: AmCalController.AmCal != null

Precondition: AmCalController.AmCal != null
- sendCalToForm()
Postcondition: Form.AmCalController == null

Procedural Behavioral Specifications of methods

+ AmCalController(Form, Loan)

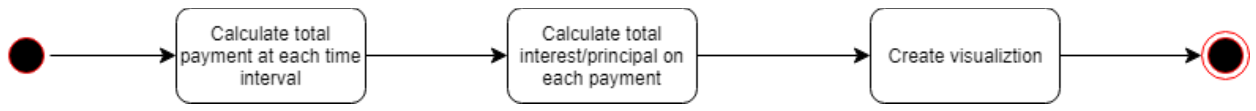


- getPayoffDate(Loan)
  this.payoffDate = Loan.payoffDate

- getInterestRate(Loan)
  this.interestRate = Loan.interestRate

- getLoanBalance(Loan)
  this.loanBalance = Loan.loanBalance

- createAmCal(Loan)

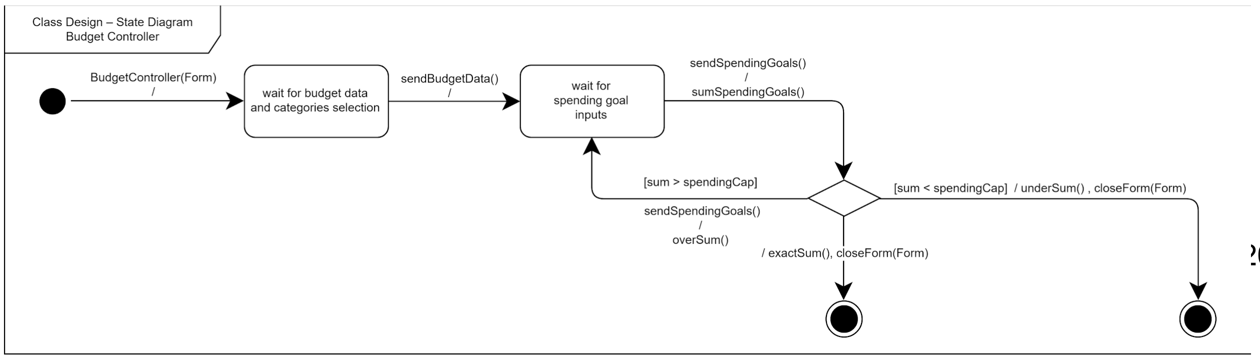

Calculate total payment at each time interval
Calculate total interest/principal on each payment
Create visualiztion

- sendCalToForm()
  this.Form.AmCal = this.AmCal

Budget Controller

OCL : context BudgetController inv: currentState < 6

Method	budgetController(Form)
Pre:	Form is not null
Post:	Set currentState to 1
Method	sendBudgetData()
Pre:	budgetStartDate, budgetEndDate, spendingCap, and categoriesList are all null
Post:	budgetStartDate, budgetEndDate, spendingCap, and categoriesList are all not null
Method	sendSpendingGoals()
Pre:	currentState < 4
Post:	spendingGoalData is not null
Method	getGoalsStatus()
Pre:	currentState > 2
Post:	goalsDone is returned
Method	overSum(budgetStart,budgetEnd,categoriesData,spendingGoalData,spendingCap)
Pre:	sum > spending cap
Post:	All elements of SpendingGoalData are set to zero
Method	underSum(budgetStart,budgetEnd,categoriesData,spendingGoalData,spendingCap)
Pre:	sum < spending cap
Post:	A category "Other" is found or concatenated and corresponding spending goal increased
Method	exactSum(budgetStart,budgetEnd,categoriesData,spendingGoalData,spendingCap)
Pre:	sum == spending cap
Post:	The budgetController is deleted

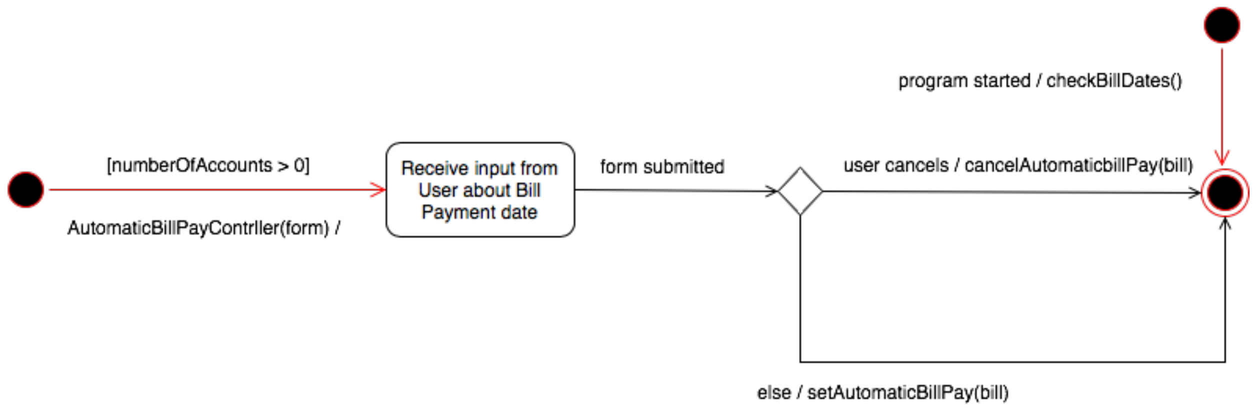

Class Design – State Diagram
Budget Controller
BudgetController(Form) /
wait for budget data and categories selection
sendBudgetData() /
wait for spending goal inputs
sendSpendingGoals() / sumSpendingGoals()
[sum > spendingCap]
sendSpendingGoals() / overSum()
[sum < spendingCap] / underSum() , closeForm(Form)
/ exactSum(), closeForm(Form)

Procedural Behavior Specification of Methods using Pseudocode:

```
+ budgetController(Form){
       currentState = 1
       goalsDone = false
}
+ sendBudgetData(){
       L = form.getInput('Input the number of categories')
       categoriesList = new String[L]
       self.budgetStartDate = form.getInput('Input the budget start date')
       self.budgetEndDate = form.getInput('Input the budget end date')
       self.spendingCap = form.getInput('Input the budget''s overall spendingCap')
       for i = 0 to L{
              self.categoriesList[i] = form.getInput('Input a category')}
       currentState = 2}
+ sendSpendingGoals(){
       self.spendingGoalData = new double[categoriesList.length]
       for i = 0 to categoriesList.length{
              self.spendingGoalData[i] = form.getInput('Input the spending goal for
this category',categoriesList[i])}
       sum = 0
       for i = 0 to spendingGoalData.length-1(){
              sum += spendingGoalData[i]}
       currentState = 3
       if sum > spendingCap
              overSum()
       else if sum < spendingCap
              underSum()
       else
              exactSum()}
+ getGoalsStatus(){
       return goalsDone}
- overSum(){
       print "Your spending goals sum to more than your spending cap! Retry."
       for i = 0 to spendingGoalData.length
              spendingGoalData[i] = 0}
- underSum(){
       goalsDone = true
       difference = spendingCap - sum

       i = 0
       done = false
       while !done{
              if categoriesList[i] == "Other" || == null
                     done = true
              else
                     i++}
       if i < categoriesList.length()
              spendingGoalData[i] = spendingGoalData[i] + difference
       else
              concatenate(categoriesList,["Other"])
              concatenate(spendingGoalData,[difference])
       currentState = 4

repository.createBudget(budgetStart,budgetEnd,categoriesList,spendingCap,spendingGoalD
ata)}
```

```
- exactSum(){
       goalsDone = true
       currentState = 5

repository.createBudget(budgetStart,budgetEnd,categoriesList,spendingCap,spendingGoalD
ata)}
```

Automatic Bill Pay Controller

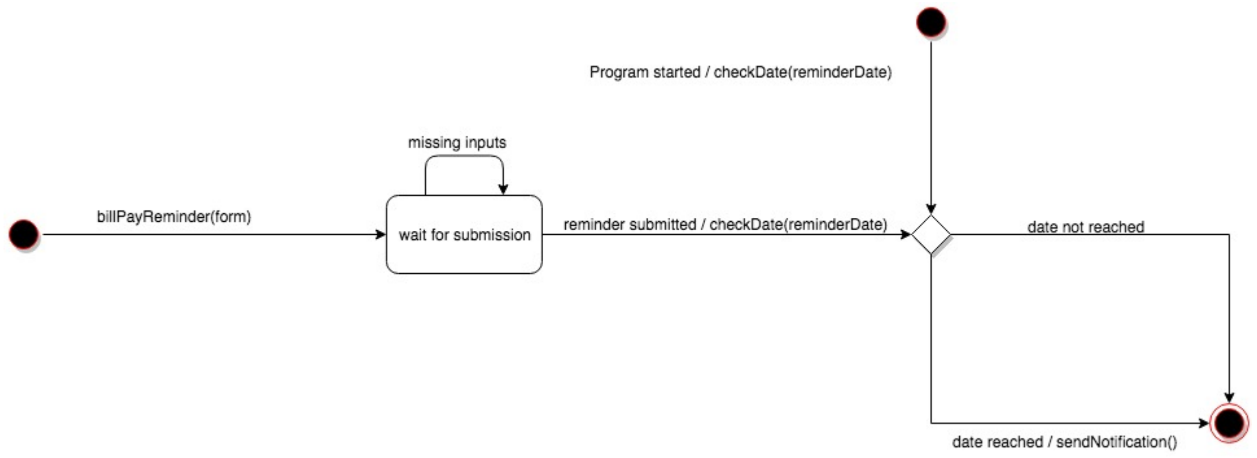

program started / checkBillDates()
[numberOfAccounts > 0]
AutomaticBillPayContrller(form) /
Receive input from User about Bill Payment date
form submitted
user cancels / cancelAutomaticbillPay(bill)
else / setAutomaticBillPay(bill)

```
Context: Automatic Bill Pay Controller Class
Invariant: numberOfAccounts > 0

Pre: Form != null
AutomaticBillPayController(Form)
Post: self != null

Pre: self.billsOnAutoPay.length > 0
cancelAutomaticBillPay(bill)
Post: bill.desiredAccount = null

Pre: bill.desiredAcount == null
setAutomaticBillPay(bill)
Post: bill.desiredAcount != null

Pre: self.bills.length > 0
checkBillDates()
Post: self.bills.length > 0

Pre: bill != null
payBill(bill)
Post: bill == null
```

Procedural Behavior Specification of Methods using Pseudocode:

```
Function AutomaticBillPayController(form)
        Set self.form equal to form

Function cancelAutomaticBillPay(bill)
        Remove bill from self.billsOnAutoPay
        Set bill equal to null

Function setAutomaticBillPay(bill)
        Set bill.desiredAccount to self.form.selectedAccount
        Add bill to self.billsOnAutoPay

Function checkBillDates()
        For i = 0 to i = self.bills.length - 1
                If billsOnAutoPay[i].duedate < Date.currentDate
                        payBill(billsOnAutoPay[i])

Function payBill(bill)
        BankInterface.sendTransacrtion(bill.name, bill.amount,
bill.desiredAccount)
        cancel AutomaticBillPay(bill)
```

Bill Pay Reminder Controller


Program started / checkDate(reminderDate)
missing inputs
billPayReminder(form)
wait for submission
reminder submitted / checkDate(reminderDate)
date not reached
date reached / sendNotification()

Class Invariant: this.Form != null && this.data != null && this.reminderDate != null

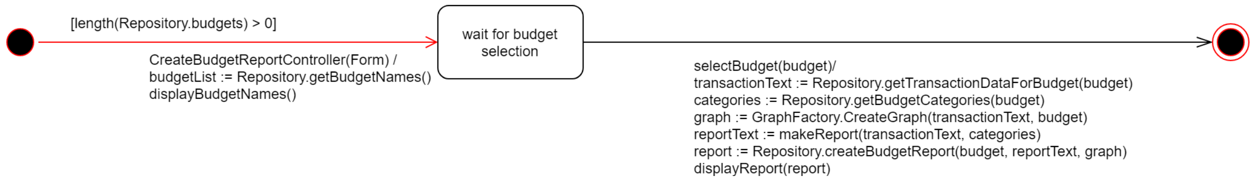
Pre/Post Conditions:
Method: billPayReminder(form)
Pre: billPayReminder.form == null
Post: billPayReminder(form)

Method: sendNotification(data)
Pre: data == null
Post: sendNotification(data)

Method: checkDate(reminderDate)
Pre: reminderDate == null
Post: checkDate(reminderDate)

```
function billPayReminder(form : Form)
      form.setTitle("New Bill Pay Reminder")
      form.add(input("Reminder Name: ", data)
      form.add(dateSelection("Reminder Date", reminderDate))

      if (checkDate(reminderDate))
            sendNotification(data)
      Repository.newBillPayReminder(data, reminderDate)

function sendNotification(data)
      showMessageDialog(frame, data)

function checkDate(reminderDate)
      if (reminderDate == Date.getDate())
            return true
      else
            return false
```

Budget Report Controller


[length(Repository.budgets) > 0]
CreateBudgetReportController(Form) /
budgetList := Repository.getBudgetNames()
displayBudgetNames()
wait for budget selection
selectBudget(budget)/
transactionText := Repository.getTransactionDataForBudget(budget)
categories := Repository.getBudgetCategories(budget)
graph := GraphFactory.CreateGraph(transactionText, budget)
reportText := makeReport(transactionText, categories)
report := Repository.createBudgetReport(budget, reportText, graph)
displayReport(report)

invariant:Repository.budgets.length > 0

Pre-condition: Repository.budgets.length > 0
Post-condition: BudgetController exists

```
CreateBudgetReportController(form: Form)
    self.form = form
    self.budgetNameList = Repository.getBudgetNames()
    displayBudgetNames()
```

Pre-condition: form is displaying budget names
Post-condition: budget report object created and displayed on form

```
selectBudget(budgetName: String)

transactionText=Repository.getTransactionDataForBudget(budgetName)
    categories = Repository.getBudgetCategories(budgetName)
    graph = GraphFactory.CreateGraph(transactionText, budgetName)
    reportText = makeReport(transactionText, categories)
    Repository.CreateBudgetReport(budgetName, reportText, graph)
    displayReport(reportText, graph)
```

pre-condition: not (form == nil) and transactionText.length > 0 and categories > 0
post-condition: reportText.length > 0

```
makeReport(transactionText[1..A]:String array, categories[1..B]: Category
array)
    reportText = ""
    ammountPerCategory[1..B] = nil
    for i = 1 to B
        for j = 1 to A
            if transactionText[j].hasText("category:" + categories[i].name)
            ammountPerCategory[i] =
             transactionText[j].getSubstring("ammount:d").getInteger()

  reportText = "Budget Report" + newLine
    for i = 1 to B
        reportText = reportText + categories[i].name
        reportText = reportText + ": $" + ammountPerCategory[i] + " of $" +
        reportText = reportText + categories.spendingGoal + newLine
        return reportText
```

Design Quality Evaluation

Our model succeeded in all stages of the detailed design part of the Design Phase of the project. First and foremost, our detailed design model excels for the fact that it is complete in all respects to the instructions given in the phase II document. Secondly, our model excels in its consistency with the diagrams made in the previous phase, with transformational accuracy of our design being of paramount importance.

Finally, we believe the design quality of our model is exceptional, as we made constant efforts to use all of the object-oriented design metrics that were at our disposal.

Our model adhered to the instructions given for phase II and is, in our view, complete. We began the detailed design aspect of phase II by making collaboration diagrams for each complex operation in our detailed system sequence diagrams (DSSD). This resulted in a total of 11 collaboration diagrams, all of which clearly state the complex operation being performed. In addition to the diagrams are GRASP descriptions which detail the various patterns used throughout our design. The controller pattern is used throughout our design, due to the fact that we have many objects that lie directly beyond the user interface layer that assist with the various system operations of our model. Another pattern used in our detailed design is the high cohesion pattern, because we wanted to make the objects our system appropriately focused, manageable, and understandable. In addition to interaction design, we successfully used our collaboration diagrams and DSSDs to develop a comprehensive design class diagram. We chose to use the repository design model because an integral part of our software involves processing data. In order to process this data efficiently, it was to our advantage to create different subsystems and connect them in a fashion that allowed them all equal access to the centralized repository. The primary type of data processing the software will do is searching through the data to see if it is relevant to the needs of the user and then formatting the data in form the user will understand. The main downside to using this format is that the repository is a very highly coupled part of the system, so maintainability will be sacrificed if there needs to be a major change to the repository layout. We also developed statecharts. In each of our state charts, for each individual state, there is a transition defined for all possible events that we could think of. Our state charts also fulfill the integral requirement of each state being targeted by at least one transition. We also made sure that in each basic state, the guards of transitions form a tautology. This ensures that our software behaves exactly how we designed it.

Our work is consistent to the best of our ability. During the analysis phase of the project, we layed out a basic framework that we have translated into our work for the design phase. We alotted time to expand on our initial work, while correcting errors and performing necessary fine-tuning as needed. Our conceptual domain model established the baseline for the design of each of our objects, and our interaction design and design class diagram we made during the design phase specified our intentions for each object by using collaboration diagrams and GRASP descriptions. Examining the consistency of our state charts, we ensured that only a single transition is triggered by a given event. After a thorough evaluation of our state charts and sequence diagrams (from the analysis phase), we also verified that every transition for a specific object was achieved by the messages sent and received by that object.This further helped in ensuring our model was consistent. Also, after examining our class diagrams, we verified that the diagram for the system describes the classes and their relationships in such a way that the behaviours specified in the sequence diagrams are correctly captured.Finally, we verified that each class is clearly defined, so as to capture the functionality specified by the state diagrams.

Using as many object-oriented design metrics as we could, we believe our model is exceptional in its quality. Examining first the simplicity of our model, we believe our design will serve the purposes of the user in a fashion that is both functional and efficient. We are confident our class design meets its objectives and has no extra or unnecessary embellishments that may bog down the operation of the software. Our control flow is complex enough to handle the operations that we intended the software to perform, while simple enough to prevent unnecessary scenarios that might frustrate the user whilst using the software. The repository architecture also allows for good information flow, considering the primary purpose of the software is to share data between subsystems. On top of simplicity, we did our best to ensure the design was modular so that different concerns of the design were separated into their respective parts. Cohesion was of utmost importance, since each component of the software will rely on other components functioning correctly. We implemented a reasonable amount of inheritance in our design, which contained more breadth than depth. This will likely come to our advantage during testing, as a model containing inheritance with more depth than breadth would take longer to test and the testing process as a whole would be more complex. In addition, we made a point to minimize the amount of coupling in our design, having enough of it to handle the large amount of data the software is intended to process, but not making so much as to over-complicate software processes. A high degree of cohesion was also important in our design in order to prevent an unnecessary amount of complexity. Another metric we used was evaluating the method complexity of our model. Many of our methods and their parameters are outlined in our collaboration diagrams and more exceedingly in our class design documents. While some methods serve more than one function, we made sure to prioritize which function each method is associated with, alleviating the issue of over-complexity. Our percentage of public variables as also very low. We make it a point to use as few public variable as possible, so as to not violate the rules of encapsulation. The number of methods in each class is enough to serve the function of the class, but not so much as to make the class overly complex. The size metrics of our design are also not unnecessarily excessive. We prioritize the number of methods for each class, and do not have an unnecessarily large number of classes.

All in all, we believe the detailed design aspect of the Design Phase of the project fulfilled our goals that we set out to achieve during the analysis phase. First, our detailed design model is complete in all respects to the instructions given in the phase II document. Secondly, our model excels in its consistency with the diagrams made in the previous phase, with transformational accuracy of our design being of paramount importance. Finally, the design quality of our model is exceptional, as we made constant efforts to use all of the object-oriented design metrics that were at our disposal.