Musa Rafik - mar6827
Lucas Best - lwb498
Jack Diao - qd572
Shawn Victor - sfv225
Kenan Hurd - kah4285

EE461L Phase 4 Report

INFORMATION HIDING

There are two main ways we implemented information hiding - utilizing model schemas and separating our backend routes into separate files. Both of these methods allowed us to maintain modularization in our design.

Model Schemas

One way we implemented information hiding is by creating schemas for each one of our MongoDB collections. This allowed us to encapsulate the data that describes each collection, such as the name, location, number of missions, etc. This is advantageous because if we need to change what we want to store or how we store it in our models, we can simply change the fields in the schemas. For example:

```
Let agencySchema = Schema{

    Name: {Type: String},

}
```

If instead of using a name to differentiate agencies, we can use id numbers. Then we can change the model like this:

```
Let agencySchema = Schema{

    Id: {Type: int}

}
```

One disadvantage of this design is that we have to handle the models differently because they have different structures. If we had a unified schema we would have less files to maintain, but because of the differences, we have to handle them each separately. This results in a more complex design. For example, we have to maintain three different types of astronauts which can lead to a lot of duplicated code simply because of how they differ in structure. One way to mitigate this problem is to maintain a consistent schema design. If we are changing what

information we want to store constantly, then we have to constantly change the schemas, whereas if we have a concrete structure, no changes should be required.

Route Files

Another way we implemented information hiding is by creating separate route files in the backend for each model request. More specifically, we had a route for launches, astronauts and agencies. The frontend just had to make a request to the specific route and our backend would handle all the work. That way, we can separate concerns such as processing the data and transforming it into a form we can use. Then the frontend just worries about rendering it in a stylistic manner. For example:

Frontend makes a fetch request for a specific astronaut:

```
fetchAstronaut = () =>{
        fetch("/UsAstronauts")
        .then(data => //show data)
}
```

In our UsAstronauts route file:

```
router.get("/", (err, res) => {
        // query database
        // process data if needed
        // send response
}
```

The frontend expects the astronaut data to come in a certain format, but it does not know how the data is stored, or how it is manipulated.

The router modularization is extremely effective at making our software extensible. For example, our three types of astronauts contain different attributes and so we created different routes for each one so that we can handle the data differently. If we add a new type of astronaut, we can

simply copy over most of the route of one of our previous astronauts and make changes to the processing data portion. Furthermore, if we want our app to utilize a router we simply include the route and add one line to our main app file:

```
app.use('/launch', launch);

app.use('/agency', agency);

…
```

We decided to use this design because we are able to work on multiple routes in parallel with no dependencies between them, such as when we were working on storing our data in our database instead of pulling directly from third-party APIs. One person could work on storing the astronaut data and another can work on launches or agencies, allowing us to speed up production.

DESIGN PATTERNS

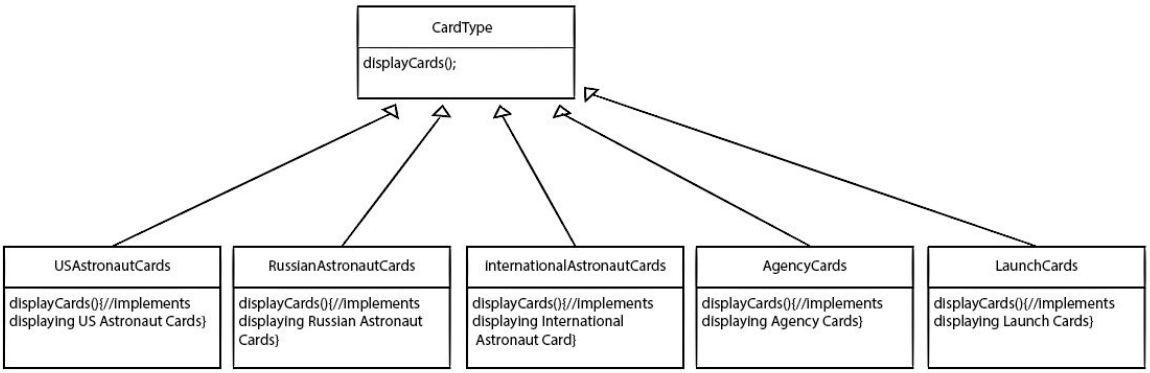
Design Pattern #1: Strategy Pattern

We decided for one implementation for a design pattern in our design would be the strategy pattern on our models pages.

Currently, when it comes to displaying all of the cards from a given model we have a single component: Cards.js which has a unique function for the process of displaying all of the cards for a specific model (GenerateUSAstronauts, GenerateRussianAstronauts, GenerateInternationalAstronauts, ect). The functions contain relatively the same algorithm for displaying except when it comes to accessing and labeling the data. Specifically the reason for this issue comes from the input table data stored within our mongodb database for each model. Not every model shares the same assessor keys. For example,the US Astronauts collection uses the key 'name' for accessing the astronaut's name. However, for the Russian Astronauts collection the key 'A' is used for accessing the astronaut name. Also the amount of data that we may want to share on all the cards of one model may not be the same for another model.

The solution to this problem would be to use the Strategy Design Pattern since it would allow us to encapsulate the different strategies within another context which will allow us to use a single function to call for generating cards, and allow us to easily swap strategies for which algorithm will be used to display the cards. Below are snippets of code showing the issue mentioned.

Before:

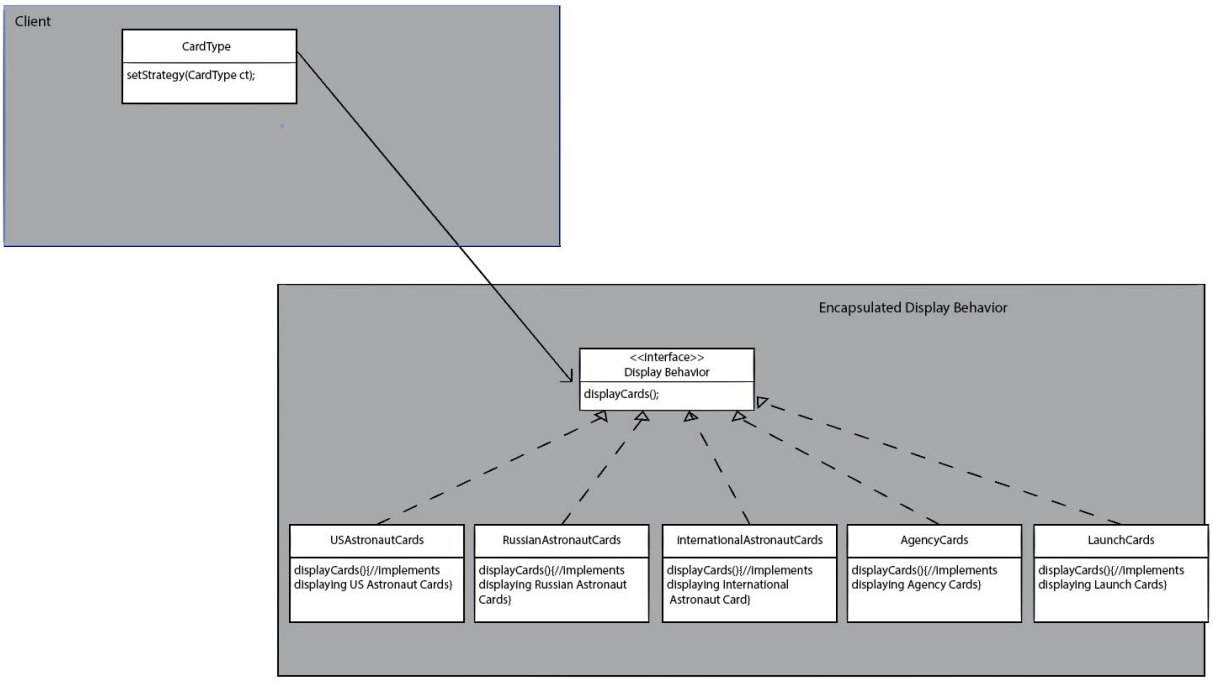



```
export default function GenerateUSAstronautCards(props){
    var counter = 0;
    return (
      <Grid>
          <div className="projects-grid" style={{display: 'flex', flex: 1, flexWrap: 'wrap'}}>
           {
              props.data.map(row => (
                <Cell col={3}>
                <Card shadow={5} style={{minWidth: '450', margin:'auto'}}>
                {(row.wikiInfo.image !== 'Not Found')
                  ? <CardTitle style={{color: '#fff', height: '350px', background:'url('+row.wikiInfo.image+') center / cover'}}>{row.Astronaut}</CardTitle>
                  : <CardTitle style={{color: '#fff', height: '350px', background:'url(https://cdn2.iconfinder.com/data/icons/russia-9/64/astronaut-avatar-profile-man-russia-512.p
                }
                    <CardActions border>
                      <CardText style={{color:'#000', textAlign:'center', fontSize:'20px'}}>
                        {row.Astronaut}
                      </CardText>
                      </CardActions>
                      <CardActions border>
                        <CardText style={{color:'#000', textAlign:'left', fontSize:'14px'}}>
                          • STATUS: {row.Status}
                          <br/>
                          • TOTAL SPACE TIME: {row['Cumulative hours of space flight time']}
                          <br/>
                          • # MISSIONS: {row['# Flights']}
                          <br/>
                          • MISSIONS: {row['Missions flown']}
                        </CardText>
                    </CardActions>
                    <CardActions border>
                      <Button onClick={event =>  window.location.href='/USAstronauts/'+row.Astronaut+'/USAstronauts'}>More Info.</Button>
                    </CardActions>
                  <CardMenu style={{color: '#fff'}}>
                    <IconButton name="share" />
                  </CardMenu>
                </Card>
              </Cell>
```

```
export function GenerateRussianAstronautCards(props)
{
  var counter = 0;
  return (
    <Grid>
        <div className="projects-grid" style={{display: 'flex', flex: 1, flexWrap: 'wrap'}}>
        {
            props.data.map(row => (
              <Cell col={3}>
                <Card shadow={5} style={{minWidth: '450', margin:'auto'}}>
                {(row.wikiInfo.image !== 'Not Found')
                  ? <CardTitle style={{color: '#fff', height: '350px', background:'url('+row.wikiInfo.image+') center / cover'}}>{row.A}</CardTitle>
                  : <CardTitle style={{color: '#fff', height: '350px', background:'url(https://cdn2.iconfinder.com/data/icons/russia-9/64/astronaut-avatar-profile-man-russia-512.
                }
                      <CardActions border>
                        <CardText style={{color:'#000', textAlign:'center', fontSize:'20px'}}>
                          {row.A}
                        </CardText>
                      </CardActions>
                      <CardActions border>
                        <CardText style={{color:'#000', textAlign:'left', fontSize:'14px'}}>
                          • STATUS: {row.C}
                          <br/>
                          • TOTAL SPACE TIME: {row.G}
                          <br/>
                          • # MISSIONS: {row.B}
                          <br/>
                          • MISSIONS: {row.F}
                        </CardText>
                      </CardActions>
                    <CardActions border>
                      <Button onClick={event =>  window.location.href='/RussianAstronauts/'+row.A+'/RussianAstronauts'}>More Info.</Button>
                    </CardActions>
                    <CardMenu style={{color: '#fff'}}>
                      <IconButton name="share" />
                    </CardMenu>
                  </Card>
                </Cell>
```

To solve this problem we will create a function inside the same class that will take in table data that could be from any model, and this function will also have a function within it that allows us to select which strategy we want to use, and also a function that allows the correct strategy to be executed using the input table data.

After:



```
export default function DisplayCards({strat, data})
{
  this.strategy = "";

  this.setStratgy = (strat)=>{
    this.strategy = strat
  }

  this.displayCard = data=>{
    return this.strategy.displayCards(data)
  }

}
```

You can see from the code snippet above an interface is created that allows for the setting of which strategy is to be used. Simply passing a reference to the desired CardType into the strat parameter allows it to be accessed and use that type as the strategy for displaying the correct set of Cards.

```
export function USAstronautCards()
{
      this.displayCards = props=>{
        return (
          <Grid>
              <div className="projects-grid" style={{display: 'flex', flex: 1, flexWrap: 'wrap'}}>
```

```
export function RussianAstronautCards()
{
      this.displayCards = props=>{
        return (
          <Grid>
              <div className="projects-grid" style={{display: 'flex', flex: 1, flexWrap: 'wrap'}}>
```

…

Each of the functions then implement their own unique strategy for the displayCards method. So by implementing this strategy we are easily able to switch between strategies and keep the code much more cleaner, concise, and easier to understand.

Design Pattern #2: Observer Pattern

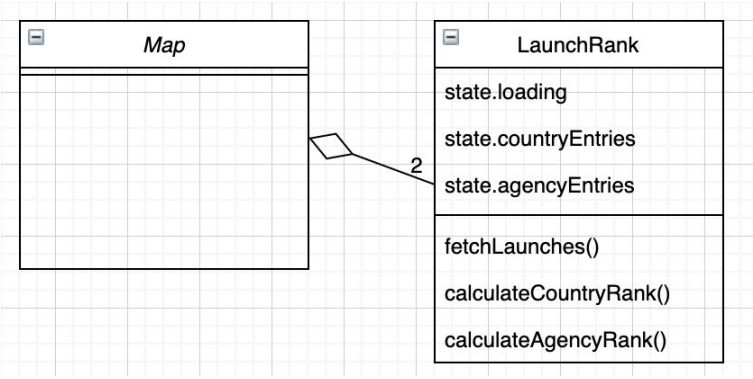
We implemented a second design pattern, the Observer Pattern to our LaunchRank class component. Currently, we have two rankings (country, agency) each requires the launch data to calculate its ranking. Before the application, they each call a fetch API to the backend to retrieve the launch data and then calculate the rank based on each launch's affiliation. We achieved the Observer Pattern by 3 steps. First, we moved the fetch API call to the Map class to pass in the launch data through props. Secondly, we add the componentDidUpdate lifecycle method in the LaunchRank class to check if the props of launch data changed. If it is changed, set loading to true and pass the launch data to the CountryRank and AgencyRank functional components. Lastly, we add these two functional components to return the correct element based on the launch data. In a nutshell, the CountryRank and AgencyRank functional components are the Observers and listen to the launch data via the props passed in from LaunchRank and the

LaunchRank notifies the observers by changing the launch data values when it changes after the fetch API call to force an update on the observer components.

The advantages: 1. There's only one fetch API call which makes the loading faster and they will be updated at the same time because they are notified at the same time once the launch data changes. 2. Each function component is only called when it is the component to render/update. 3. It is easier to add more similar ranking components later. The disadvantage: there are more components, the code is potentially longer.

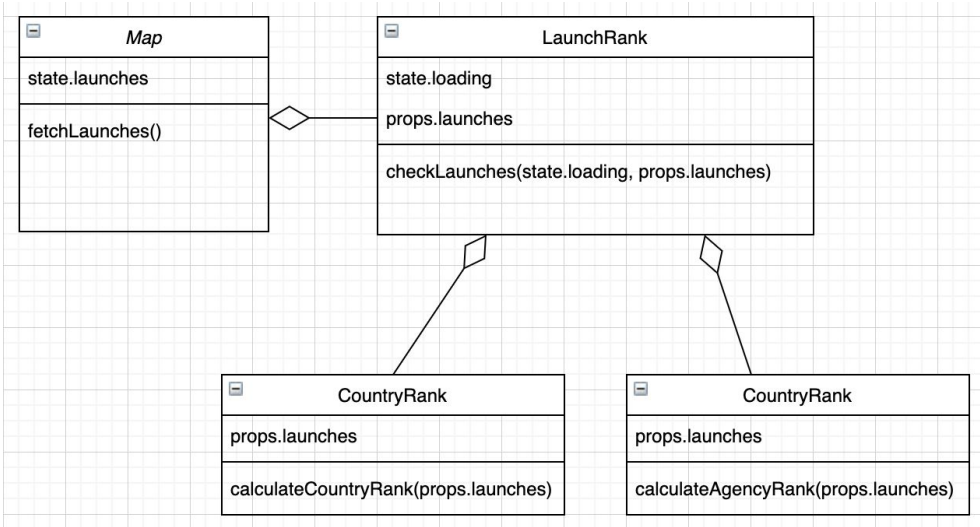
Before:

```
class Map extends Component {
    render() {
        return <div><LaunchRank category='country'/><LaunchRank category='agency'/></div>;
    }
}
class LaunchRank extends Component {
    constructor(props) {
        super(props);
        this.state = {
          loading: true,
          countryEntries: [],
          agencyEntries: []
        }
    }
    componentDidMount() {
        fetchLaunches()
        .then((launches) => {
            this.setState({
                countryEntries: calculateCountryRank(launches),
                agencyEntries: calculateAgencyRank(launches)
            });
        });
    }
    render() {
        return( this.props.category === 'country'
            ? <React.Fragment>{/*Country rank using data from this.state.countryEntries*/}</React.Fragment>
            : <React.Fragment>{/*Agency rank using data from this.state.agencyEntries*/}</React.Fragment>
        )
    }
}
```



After:

```
class Map extends Component {
    constructor(){
        this.state = { launches: [] }
    }
    componentDidMount() {
        fetchLaunches()
        .then(data => { this.setState({launches: data}) });
    }
    render() {
        return <div><LaunchRank category='country' launches={this.state.launches}/><LaunchRank category='agency' launches={this.state.launches}/></div>;
    }
}
class LaunchRank extends Component {
    constructor(props) {
        super(props);
        this.state = { loading: true }
    }
    componentDidUpdate() {
        if(this.props.launches.length > 0 && this.state.loading)
            this.setState({ loading: false });
    }
    render() {
        switch (this.props.category) {
            case 'country':
                return <CountryRank loading={this.state.loading} launches={this.props.launches} />;
            case 'agency':
                return <AgencyRank loading={this.state.loading} launches={this.props.launches} />;
        }
    }
}
const CountryRank = (props) => {
    const countryEntries = calculateCountryRank(props.launches);
    return <React.Fragment>{/*Country rank using data from countryEntries or loading message if props.loading === true*/}</React.Fragment>
}
const AgencyRank = (props) => {
    const agencyEntries = calculateAgencyRank(props.launches);
    return <React.Fragment>{/*Agency rank using data from agencyEntries or loading message if props.loading === true*/}</React.Fragment>
}
```



REFACTORING

Here are three examples of refactoring for this phase. We did two extract method refactorings and one pull up method refactoring

Example 1: Extract Method

In our backend we had a function that would query a Wikipedia API to get information about our models (in this case agencies). The first code snippet shows our initial function where all the logic of how to create the updated document with the new data as well as adding it to MongoDB are placed in the same function. To make the code clearer to understand and more organized, we extracted the logic to create the new updated document into a separate function called createWikiObject(). The second and third code snippets show the new function as well as the change to the original function. In the last snippet, one can see that all the logic to get the Wikipedia information is contained in just one function call. Now, the function does not need to know how the document is updated, just that it has to add it to MongoDB.

Before:

```
getWikiInfo = (agency) =>{
    let searchTerm = '';
    if(agency.wikiURL){
        let temp = agency.wikiURL.split('/');
        searchTerm = temp[temp.length - 1];
    }
    else{
        searchTerm = agency.name;
    }
    let url = "http://en.wikipedia.org/api/rest_v1/page/summary/" + searchTerm;
    request(url, (req, response) =>{
        let results = JSON.parse(response.body);
        if(results.title !== 'Not found.'){
            let object = '';
            if(results.thumbnail){
                object = {'title': results.title, 'page': results.content_urls.desktop.page, 'extract': results.extract, 'image
            }
            else{
                object = {'title': results.title, 'page': results.content_urls.desktop.page, 'extract': results.extract, 'image
            }
            agency.wikiInfo = object;
        }
        else{
            let object = {'title': 'Not found', 'page': 'Not found', 'extract': 'Not found', 'image': 'Not found'};
            agency.wikiInfo = object;
        }
```

After:

```
createWikiObject = (results) =>{
    let wikiObject = '';
    if(results.title !== 'Not found.'){
        if(results.thumbnail){
            wikiObject = {'title': results.title, 'page': results.content_urls.desktop.page, 'extract': results.extract,
        }
        else{
            wikiObject = {'title': results.title, 'page': results.content_urls.desktop.page, 'extract': results.extract,
        }
    }
    else{
        wikiObject = {'title': 'Not found', 'page': 'Not found', 'extract': 'Not found', 'image': 'Not found'};
    }
    return wikiObject
}
```

```
// Send agency name into wikipedia request and add result to
getWikiInfo = (agency) =>{
    let searchTerm = '';
    if(agency.wikiURL){
        let temp = agency.wikiURL.split('/');
        searchTerm = temp[temp.length - 1];
    }
    else{
        searchTerm = agency.name;
    }
    let url = "http://en.wikipedia.org/api/rest_v1/page/summ
    request(url, (req, response) =>{
        let results = JSON.parse(response.body);
        agency.wikiInfo = createWikiObject(results);
        Agency.create(agency, (err, result) =>{
            if(err){
                console.log(err);
            }
            else{
```

Example 2: Extract Method

Before storing our astronauts in MongoDB, we had to standardize their names to work with our function to get Wikipedia information, as well as to look stylistically better on our frontend. We had one function that parsed the astronauts name by first cleaning up any punctuation and then determining what the first, middle and last names were. We extracted the cleaning up portion and created a new function for that task. This is useful because we can extend the function to clean up more things than just periods and commas without muddying up the overall parsing function. The following code snippets show the extraction and how the parse function just has to call the cleaning function.

Before:

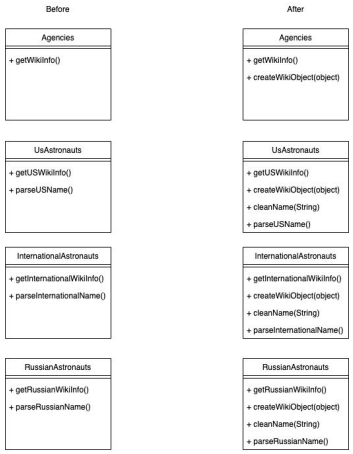
```
// convert name into firstname, lastname
parseUSName = (name) =>{
    let nameArray = name.split(" ");
    for(let i = 0; i < nameArray.length; i++){
        nameArray[i] = nameArray[i].replace(",", "");
        nameArray[i] = nameArray[i].replace(".", "");
    }
```

After:

```
cleanName = (name) =>{
    let cleanNameArray = name.split(" ");
    for(let i = 0; i < cleanNameArray.length; i++){
        cleanNameArray[i] = cleanNameArray[i].replace(",", "");
        cleanNameArray[i] = cleanNameArray[i].replace(".", "");
    }
}

// convert name into firstname, lastname
parseUSName = (name) =>{
    let cleanNameArray = cleanName(name);
```

UML Diagram with both extract method refactorings:



Example 3: Pull Up Method

For each of the astronaut routes in our backend, there exists a parse method for creating a clean and consistent name for each astronaut, as discussed in Refactor Example 2. This method is the exact same in USAstronautsRoute.js as well as RussianAstronautsRoute.js, and is only slightly altered in InternationalAstronautsRoute.js. In order to reduce the amount of code, as well as ensure that code is combined and consistent when possible, we used the pull up method on the parse and cleanup functions in these classes. Doing so meant adapting the methods to all match each other and then moving these equivalent methods to a superclass that all 3 of the aforementioned classes would access the method from. The following code segments show the methods before combination/movement as well as after.

Before:

USAstronautsRoute.js

```
cleanName = (name) =>{
    let cleanNameArray = name.split(" ");
    for(let i = 0; i < cleanNameArray.length; i++){
        cleanNameArray[i] = cleanNameArray[i].replace(",", "");
        cleanNameArray[i] = cleanNameArray[i].replace(".", "");
    }
}

// convert nase into firstname, lastnase
parseUSName = (name) =>{
    let cleanNameArray = cleanName(name);

    // Take out blank spaces and initials since they mess up formation of full name
    let finalNameArray = []
    for(let i = 0; i < cleanNameArray.length; i++){
        if(cleanNameArray[i] !== '' && cleanNameArray[i].length > 1){
            finalNameArray.push(cleanNameArray[i]);
        }
    }

    let firstName = finalNameArray[1];
    let lastName = finalNameArray[0];
    let fullName  = '';
    if(finalNameArray.length > 2){
        let middleName = finalNameArray[2];
        fullName = firstName + ' ' + middleName + ' ' + lastName; // Might not need middle name
    }
    else{
        fullName = firstName + ' ' + lastName;
    }
    fullName = firstName + ' ' + lastName;
    return fullName;
}
```

RussianAstronautsRoute.js

```
cleanName = (name) =>{
    let cleanNameArray = name.split(" ");
    for(let i = 0; i < cleanNameArray.length; i++){
        cleanNameArray[i] = cleanNameArray[i].replace(",", "");
        cleanNameArray[i] = cleanNameArray[i].replace(".", "");
    }
}

// Convert name to first name, last name
parseRussianName = (name) =>{
    let cleanNameArray = cleanName(name);

    // Take out blank spaces and initials since they mess up formation of full name
    let finalNameArray = []
    for(let i = 0; i < cleanNameArray.length; i++){
        if(cleanNameArray[i] !== '' && cleanNameArray[i].length > 1){
            finalNameArray.push(cleanNameArray[i]);
        }
    }

    let firstName = finalNameArray[1];
    let lastName = finalNameArray[0];
    let fullName  = '';
    if(finalNameArray.length > 2){
        let middleName = finalNameArray[2];
        fullName = firstName + ' ' + middleName + ' ' + lastName; // Might not need middle name
    }
    else{
        fullName = firstName + ' ' + lastName;
    }
    fullName = firstName + ' ' + lastName;
    return fullName;
}
```

InternationalAstronautsRoute.js

```
cleanName = (name) =>{
    let cleanNameArray = name.split(" ");
    for(let i = 0; i < cleanNameArray.length; i++){
        cleanNameArray[i] = cleanNameArray[i].replace(",", "");
        cleanNameArray[i] = cleanNameArray[i].replace(".", "");
    }
}

parseInternationalName = (name) =>{
    let cleanNameArray = cleanName(name);

    // Take out blank spaces and initials since they mess up formation of full name
    let finalNameArray = []
    for(let i = 0; i < cleanNameArray.length; i++){
        if(cleanNameArray[i] !== '' && cleanNameArray[i].length > 1){
            finalNameArray.push(cleanNameArray[i]);
        }
    }

    // order of names in array is expected to be [middle name, last name, first name]
    let firstName = finalNameArray[finalNameArray.length - 1];
    let lastName = finalNameArray[finalNameArray.length - 2];
    let middleNames = []; // An array for names with multiple middle names
    if(finalNameArray.length > 2){
        middleNames = finalNameArray.slice(0, cleanNameArray.length - 2);
    }
    let middleName = ' ';
    for(let i = 0; i < middleNames.length; i++){ // Combine middle names into one string
        if(middleNames[i].length > 1){
            middleName += middleNames[i];
        }
    }

    return firstName + middleName + ' ' + lastName; // Create full name to pass into request
}
```

After:

After refactoring, the clean and parse methods are removed from individual files and all relocated to parseAstronautNames.js

parseAstronautNames.js

```
cleanName = (name) =>{
    let cleanNameArray = name.split(" ");
    for(let i = 0; i < cleanNameArray.length; i++){
        cleanNameArray[i] = cleanNameArray[i].replace(",", "");
        cleanNameArray[i] = cleanNameArray[i].replace(".", "");
    }
}

parseAstronautName = (name) =>{
    let cleanNameArray = cleanName(name);

    // Take out blank spaces and initials since they mess up formation of full name
    let finalNameArray = []
    for(let i = 0; i < cleanNameArray.length; i++){
        if(cleanNameArray[i] !== '' && cleanNameArray[i].length > 1){
            finalNameArray.push(cleanNameArray[i]);
        }
    }

    let firstName = finalNameArray[1];
    let lastName = finalNameArray[0];
    let fullName  = '';
    if(finalNameArray.length > 2){
        let middleName = finalNameArray[2];
        fullName = firstName + ' ' + middleName + ' ' + lastName; // Might not need middle name
    }
    else{
        fullName = firstName + ' ' + lastName;
    }
    fullName = firstName + ' ' + lastName;
    return fullName;
}

module.exports.parseAstronautName = parseAstronautName;
```

All 3 astronaut classes that used this method will now simply require the method and then use it as follows.

```
var parseName = require('./parseAstronautNames.js');
```

```
let convertedName = parseName(responseArray[i].Astronaut);
```

UML Diagram of Pull Method Refactoring:

Before:

After: