Software Engineering Lab 7

Name: Mausam Kamdar
ID: 202201372

Q1
CODE
GitHub link: modulo/src/Modulo.js at main · modulojs/modulo

```
// Copyright 2024 MichaelB | https://modulojs.org | Modulo v0.0.72 | LGPLv3
// Modulo LGPLv3 NOTICE: Any direct modifications to the Modulo.js source code
// must be LGPL or compatible. It is acceptable to distribute dissimilarly
// licensed code built with the Modulo framework bundled in the same file for
// efficiency instead of "linking", as long as this notice and license remains
// intact with the Modulo.js source code itself and any direct modifications.
if (typeof window === "undefined") { // Allow for Node.js environment
    var window = {};
}
window.ModuloPrevious = window.Modulo;
window.moduloPrevious = window.modulo;
window.Modulo = class Modulo {
    constructor() {
        window._moduloID = (window._moduloID || 0) + 1;
        this.window = window;
        this.id = window._moduloID; // Every Modulo instance gets a unique ID.
        this._configSteps = 0; // Used to check for an infinite loop during load
        this.registry = { cparts: { }, coreDefs: { }, utils: { }, core: { },
                          engines: { }, commands: { }, templateFilters: { },
                          templateTags: { }, processors: { }, elements: { } };
        this.config = {}; // Default confs for classes (e.g. all Components)
        this.definitions = {}; // For specific definitions (e.g. one Component)
        this.stores = {}; // Global data store (by default, only used by State)
    }

    register(type, cls, defaults = undefined) {
        type = (`${type}s` in this.registry) ? `${type}s` : type; // pluralize
        if (type in this.registry.registryCallbacks) {
            cls = this.registry.registryCallbacks[type](this,  cls) || cls;
        }
        this.assert(type in this.registry, 'Unknown registry type: ' + type);
        this.registry[type][cls.name] = cls;
        if (cls.name[0].toUpperCase() === cls.name[0]) { // e.g. class FooBar
            const conf = this.config[cls.name.toLowerCase()] || {};
            Object.assign(conf, { Type: cls.name }, cls.defaults, defaults);
            this.config[cls.name.toLowerCase()] = conf; // e.g. config.foobar
        }
    }

    instance(def, extra, inst = null) {
        const isLower = key => key[0].toLowerCase() === key[0];
        const coreDefSet = { Component: 1, Artifact: 1 }; // TODO: make compatible with any
registration type
        const registry = (def.Type in coreDefSet) ? 'coreDefs' : 'cparts';
        inst = inst || new this.registry[registry][def.Type](this, def, extra.element || null); // TODO rm
the element arg
        const id = ++window._moduloID;
        //const conf = Object.assign({}, this.config[name.toLowerCase()], def);
        const conf = Object.assign({}, def); // Just shallow copy "def"
        const attrs = this.registry.utils.keyFilter(conf, isLower);
        Object.assign(inst, { id, attrs, conf }, extra, { modulo: this });
        if (inst.constructedCallback) {
            inst.constructedCallback();
        }
        return inst;
    }

    instanceParts(def, extra, parts = {}) {
        // Loop through all children, instancing each class with configuration
        const allNames = [ def.DefinitionName ].concat(def.ChildrenNames);
        for (const def of allNames.map(name => this.definitions[name])) {
            parts[def.RenderObj || def.Name] = this.instance(def, extra);
        }
        return parts;
    }

    lifecycle(parts, renderObj, lifecycleNames) {
        for (const lifecycleName of lifecycleNames) {
            const methodName = lifecycleName + 'Callback';
            for (const [ name, obj ] of Object.entries(parts)) {
                if (!(methodName in obj)) {
                    continue; // Skip if obj has not registered callback
                }
                const result = obj[methodName].call(obj, renderObj);
                if (result) { // TODO: Change to (result !== undefined) and test
                    renderObj[obj.conf.RenderObj || obj.conf.Name] = result;
                }
            }
        }
    }
    preprocessAndDefine(cb, prefix = 'Def') {
        this.fetchQueue.enqueue(() => {
            this.repeatProcessors(null, prefix + 'Builders', () => {
                this.repeatProcessors(null, prefix + 'Finalizers', cb || (() => {}));
            });
        }, true); // The "true" causes it to wait for all
    }

    loadString(text, parentName = null) { // TODO: Refactor this method away
        return this.loadFromDOM(this.registry.utils.newNode(text), parentName);
    }

    loadFromDOM(elem, parentName = null, quietErrors = false) { // TODO: Refactor this method
away
        const loader = new this.registry.core.DOMLoader(this);
        return loader.loadFromDOM(elem, parentName, quietErrors);
    }

    repeatProcessors(defs, field, cb) {
        let changed = true; // Run at least once
        const defaults = this.config.modulo['default' + field] || [];
        while (changed) {
            this.assert(this._configSteps++ < 90000, 'Config steps: 90000+');
            changed = false; // TODO: Is values deterministic in order? (Solution, if necessary:
definitions key order arr)
            for (const def of (defs || Object.values(this.definitions))) {
                const processors = def[field] || defaults;
                //changed = changed || this.applyProcessors(def, processors);
                const result = this.applyNextProcessor(def, processors);
                if (result === 'wait') { // TODO: Refactor logic here &
                    changed = null; // null always triggers an enqueue
                    break;
                }
                changed = changed || result;
            }
        }
        const repeat = () => this.repeatProcessors(defs, field, cb);
        if (changed !== null && Object.keys(this.fetchQueue ? this.fetchQueue.queue : {}).length
=== 0) { // TODO: Remove ?: after core object refactor
            if (cb) {
                cb(); // Synchronous path
            }
        } else {
            this.fetchQueue.enqueue(repeat);
        }
    }

    applyNextProcessor(def, processorNameArray) {
        const cls = this.registry.cparts[def.Type] || this.registry.coreDefs[def.Type] || {}; // TODO: Fix
this
        const { processors } = this.registry;
        for (const name of processorNameArray) {
            const [ attrName, aliasedName ] = name.split('|');
            if (attrName in def) {
                const funcName = aliasedName || attrName;
                const proc = this.registry.processors[funcName.toLowerCase()];
                const func = funcName in cls ? cls[funcName].bind(cls) : proc;
                const value = def[attrName]; // Pluck value & remove attribute
                delete def[attrName]; // TODO: document 'wait' or rm -v
                return func(this, def, value) === true ? 'wait' : true;
            }
        }
        return false; // No processors were applied, return false
    }

    assert(value, ...info) {
        if (!value) {
            console.error(this.id, ...info);
            throw new Error(`Modulo Error: "${Array.from(info).join(' ')}"`);
        }
    }
}

// TODO: Move to conf
window.Modulo.INVALID_WORDS = new Set((`
    break case catch class const continue debugger default delete do else enum
    export extends finally for if implements import in instanceof interface new
    null package private protected public return static super switch throw try
    typeof var let void  while with await async true false
`).split(/\s+/ig));

// Create a new modulo instance to be the global default instance
window.modulo = new window.Modulo();
if (typeof modulo === "undefined" || modulo.id !== window.modulo.id) {
    var modulo = window.modulo; // TODO: RM when global modulo is cleaned up
}
window.modulo.registry.registryCallbacks = { // Set up default registry hooks
    commands(modulo, cls) {
        window.m = window.m || {}; // Avoid overwriting existing truthy m
        window.m[cls.name] = () => cls(modulo); // Attach shortcut to global "m"
    },
    processors(modulo, cls) {
        modulo.registry.processors[cls.name.toLowerCase()] = cls; // Alias lower
    },
    core(modulo, cls) { // Global / core class getting registered
        const lowerName = cls.name[0].toLowerCase() + cls.name.slice(1);
        modulo[lowerName] = new cls(modulo);
        modulo.assets = modulo.assetManager; // TODO Rm
    },
};

modulo.register('coreDef', class Modulo {}, {
    ChildPrefix: '', // Prevents all children from getting modulo_ prefixed
    Contains: 'coreDefs',
    DefLoaders: [ 'DefTarget', 'DefinedAs', 'Src', 'Content' ],
    defaultDef: { DefTarget: null, DefinedAs: null, DefName: null },
    defaultDefLoaders: [ 'DefTarget', 'DefinedAs', 'Src' ],
    defaultFactory: [ 'RenderObj', 'factoryCallback' ], // TODO: this might be dead code?
});

modulo.register('core', class ValueResolver {
    constructor(contextObj = null) {
        this.ctxObj = contextObj;
        this.isJSON = /^(true$|false$|null$|[^a-zA-Z])/; // "If not variable"
    }

    get(key, ctxObj = null) {
        const { get } = window.modulo.registry.utils; // For drilling down "."
        const obj = ctxObj || this.ctxObj; // Use given one or in general
        return this.isJSON.test(key) ? JSON.parse(key) : get(obj, key);
    }

    set(obj, keyPath, val, autoBind = false) {
        const index = keyPath.lastIndexOf('.') + 1; // Index at 1 (0 if missing)
        const key = keyPath.slice(index).replace(/:$/, ''); // Between "." & ":"
        const prefix = keyPath.slice(0, index - 1); // Get before first "."
        const target = index ? this.get(prefix, obj) : obj; // Drill down prefix

        if (keyPath.endsWith(':')) { // If it's a dataProp style attribute
            const parentKey = val.substr(0, val.lastIndexOf('.'));
            val = this.get(val); // Resolve "val" from context, or JSON literal
            if (autoBind && !this.isJSON.test(val) && parentKey.includes('.')) {
```

```
            val = val.bind(this.get(parentKey)); // Parent is sub-obj, bind
          }
        }
        target[key] = val; // Assign the value to it's parent object
    }
});

modulo.register('core', class DOMLoader {
    constructor(modulo) {
        this.modulo = modulo; // TODO: need to standardize back references to prevent
mismatches
    }

    getAllowedChildTags(parentName) {
        let tagsLower = this.modulo.config.domloader.topLevelTags; // "Modulo"
        if (/^_[a-z][a-zA-Z]+$/.test(parentName)) { // _likethis, e.g. _artifact
            tagsLower = [ parentName.toLowerCase().replace('_', '') ]; // Dead code?
        } else if (parentName) { // Normal parent, e.g. Library, Component etc
            const parentDef = this.modulo.definitions[parentName];
            const msg = `Invalid parent: ${ parentName } (${ parentDef })`;
            this.modulo.assert(parentDef && parentDef.Contains, msg);
            const names = Object.keys(this.modulo.registry[parentDef.Contains]);
            tagsLower = names.map(s => s.toLowerCase()); // Ignore case
        }
        return tagsLower;
    }

    loadFromDOM(elem, parentName = null, quietErrors = false) {
        const { defaultDef } = this.modulo.config.modulo;
        const toCamel = s => s.replace(/-([a-z])/g, g => g[1].toUpperCase());
        const tagsLower = this.getAllowedChildTags(parentName);
        const array = [];
        for (const node of elem.children || []) {
            const partTypeLC = this.getDefType(node, tagsLower, quietErrors);
            if (node._moduloLoadedBy || partTypeLC === null) {
                continue; // Already loaded, or an ignorable or silenced error
            }
            node._moduloLoadedBy = this.modulo.id; // Mark as having loaded this
            // Valid definition, now create the "def" object
            const def = Object.assign({ Parent: parentName }, defaultDef);
            def.Content = node.tagName === 'SCRIPT' ? node.textContent : node.innerHTML;
            array.push(Object.assign(def, this.modulo.config[partTypeLC]));
            for (let name of node.getAttributeNames()) { // Loop through attrs
                const value = node.getAttribute(name);
                if (partTypeLC === name && !value) { // e.g. <def Script>
                    continue; // This is the "Type" attribute itself, skip
                }
                def[toCamel(name)] = value; // "-kebab-case" to "CamelCase"
            }
        }
        this.modulo.repeatProcessors(array, 'DefLoaders');
        return array;
    }

    getDefType(node, tagsLower, quiet = false) {
        const { tagName, nodeType, textContent } = node;
        if (nodeType !== 1) { // Text nodes, comment nodes, etc
            if (nodeType === 3 && textContent && textContent.trim() && !quiet) {
                console.error(`Unexpected text in definition: ${textContent}`);
            }
            return null;
        }
        let defType = tagName.toLowerCase();
        if (defType in this.modulo.config.domloader.genericDefTags) {
            for (const attrUnknownCase of node.getAttributeNames()) {
                const attr = attrUnknownCase.toLowerCase();
                if (!node.getAttribute(attr) && tagsLower.includes(attr)) {
                    defType = attr; // Has an empty string value, is a def
                }
                break; // Always break: We will only look at first attribute
            }
        }
        if (!(tagsLower.includes(defType))) { // Were any discovered?
            if (defType === 'testsuite') { return null; } /* TODO Remove and add recipe to stub /
silence TestSuite not found errors */
            if (!quiet) { // Invalid def / cPart: This type is not allowed here
                console.error(`"${ defType }" is not one of: ${ tagsLower }`);
            }
            return null // Return null to signify not a definition
        }
        return defType; // Valid, expected definition: Return lowercase type
    }
}, {
    topLevelTags: [ 'modulo' ], // Only "Modulo" is top
    genericDefTags: { def: 1, script: 1, template: 1, style: 1 },
});

modulo.register('processor', function src (modulo, def, value) {
    const { getParentDefPath } = modulo.registry.utils;
    def.Source = (new window.URL(value, getParentDefPath(modulo, def))).href;
    modulo.fetchQueue.fetch(def.Source).then(text => {
        def.Content = (text || '') + (def.Content || '');
    });
});

modulo.register('processor', function srcSync (modulo, def, value) {
    modulo.registry.processors.src(modulo, def, value);
    return true; // Only difference is return "true" for "wait" (TODO: Refactor to "return def.SrcSync
? false" then specify on Configuration)
});

modulo.register('processor', function defTarget (modulo, def, value) {
    const resolverName = def.DefResolver || 'ValueResolver'; // TODO: document
    const resolver = new modulo.registry.core[resolverName](modulo);
    const target = value === null ? def : resolver.get(value); // Target object
    for (const [ key, defValue ] of Object.entries(def)) { // Resolve all values
        if (key.endsWith(':') || key.includes('.')) {
            delete def[key]; // Remove & replace unresolved value
            //resolver.set(/[^a-z]/.test(key) ? target : def, key, defValue); // TODO: Probably should be
this -- not sure how this interacts with if
            resolver.set(/^[a-z]/.test(key) ? target : def, key, defValue);
        }
    }
});

modulo.register('processor', function content (modulo, conf, value) {
    modulo.loadString(value, conf.DefinitionName);
});

modulo.register('processor', function definedAs (modulo, def, value) {
    def.Name = value ? def[value] : (def.Name || def.Type.toLowerCase());
    const parentDef = modulo.definitions[def.Parent];
    const parentPrefix = parentDef && ('ChildPrefix' in parentDef) ?
        parentDef.ChildPrefix : (def.Parent ? def.Parent + '_' : '');
    def.DefinitionName = parentPrefix + def.Name;
    // Search for the next free Name by suffixing numbers
    while (def.DefinitionName in modulo.definitions) {
        const match = /([0-9]+)$/.exec(def.Name);
        const number = match ? match[0] : '';
        def.Name = def.Name.replace(number, '') + ((number * 1) + 1);
        def.DefinitionName = parentPrefix + def.Name;
    }
```

```
        modulo.definitions[def.DefinitionName] = def; // store definition
        const parentConf = modulo.definitions[def.Parent];
        if (parentConf) {
            parentConf.ChildrenNames = parentConf.ChildrenNames || [];
            parentConf.ChildrenNames.push(def.DefinitionName);
        }
    });

    modulo.register('util', function initComponentClass (modulo, def, cls) {
        // Run factoryCallback static lifecycle method to create initRenderObj
        const initRenderObj = { elementClass: cls };
        for (const defName of def.ChildrenNames) {
            const cpartDef = modulo.definitions[defName];
            const cpartCls = modulo.registry.cparts[cpartDef.Type];
            if (cpartCls.factoryCallback) {
                const result = cpartCls.factoryCallback(initRenderObj, cpartDef, modulo);
                initRenderObj[cpartDef.RenderObj || cpartDef.Name] = result;
            }
        }

        cls.prototype.init = function init () {
            this.modulo = modulo;
            this.isMounted = false;
            this.isModulo = true;
            this.originalHTML = null;
            this.originalChildren = [];
            this.cparts = modulo.instanceParts(def, { element: this });
        };
        modulo._connectedQueue = modulo._connectedQueue || []; // Ensure array
        modulo._drainQueue = () => { // "Clusters" all moduloMount calls
            while (modulo._connectedQueue.length > 0) { // Drains + invokes
                modulo._connectedQueue.shift().moduloMount();
            }
        };
        cls.prototype.connectedCallback = function connectedCallback () {
            modulo._connectedQueue.push(this);
            window.setTimeout(modulo._drainQueue, 0);
        };
        cls.prototype.moduloMount = function moduloMount(force = false) {
            if ((!this.isMounted && window.document.contains(this)) || force) {
                this.cparts.component._lifecycle([ 'initialized', 'mount', 'mountRender' ]);
            }
        };
        cls.prototype.initRenderObj = initRenderObj;
        cls.prototype.rerender = function (original = null) {
            if (!this.isMounted) { // Not mounted, do Mount which will also rerender
                return this.moduloMount();
            }
            this.cparts.component.rerender(original); // Otherwise, normal rerender
        };
        cls.prototype.getCurrentRenderObj = function () {
            return this.cparts.component.getCurrentRenderObj();
        };
        modulo.register('element', cls); // All elements get registered centrally
    });

    modulo.register('util', function makeStore (modulo, def) {
        const isLower = key => key[0].toLowerCase() === key[0]; // skip "-prefixed"
        let data = modulo.registry.utils.keyFilter(def, isLower); // Get defaults
        data = JSON.parse(JSON.stringify(data)); // Deep copy to ensure primitives
        return { data, boundElements: {}, subscribers: [] };
    });

    modulo.register('processor', function mainRequire (modulo, conf, value) {
        modulo.assets.mainRequire(value);
    });

    modulo.config.artifact = {
        BuildCommandFinalizers: [ 'SaveArtifact' ],
        SaveArtifact: null,
        DefinedAs: 'name',
        exclude: '[modulo-asset]',
    };
    modulo.register('coreDef', class Artifact {
        static SaveArtifact (modulo, def, value) { // Build processor
            const artifactInstance = modulo.instance(def, { });
            artifactInstance.buildCommand(document); // e is document or target
            return true; // Always wait for next
        }

        getBundle(targetElem) { // TODO: Mix in targetElem to QSA
            const bundledElems = [];
            for (const elem of targetElem.querySelectorAll(this.conf.bundle)) {
                const url = elem.getAttribute('src') || elem.getAttribute('href');
                if (this.conf.exclude && elem.matches(this.conf.exclude) || !url) {
                    continue; // Need skip, otherwise it chokes assets or blank src
                }
                this.modulo.fetchQueue.fetch(url).then(text => { // Enqueue fetch
                    delete this.modulo.fetchQueue.data[url]; // remove from cache
                    elem.bundledContent = text; // attach back to element for later
                });
                bundledElems.push(elem);
            }
            return bundledElems;
        }
        getTemplateContext(targetElem) {
            const head = targetElem.head || { innerHTML: '' };
            const body = targetElem.body || { innerHTML: '', id: '' };
            const htmlPrefix = '<!DOCTYPE html><html><head>' + head.innerHTML;
            const htmlInterfix = '</head><body id="' + body.id + '">' + body.innerHTML;
            const htmlSuffix = '</body></html>';
            const bundle = this.conf.bundle ? this.getBundle(targetElem) : null;
            const extras = { htmlPrefix, htmlInterfix, htmlSuffix, bundle };
            return Object.assign({ }, this.modulo, extras);
        }

        buildCommand(targetElem) {
            const { Template } = this.modulo.registry.cparts;
            const { saveFileAs, hash } = this.modulo.registry.utils;
            const { DefinitionName, remove, urlName, name, Content } = this.conf;
            const def = this.modulo.definitions[DefinitionName]; // Get original def
            if (remove) { // Need to remove elements from document first
                targetElem.querySelectorAll(remove).forEach(elem => elem.remove());
            }
            this.templateContext = this.getTemplateContext(targetElem); // Queue up
            this.modulo.fetchQueue.enqueue(() => { // Drain queue before continue
                const tmplt = new Template(Content); // Render file Artifact content
                const content = tmplt.render(this.templateContext);
                def.FileName = `modulo-build-${ hash(content) }.${ name }`;
                if (urlName) { // Guess filename based on URL (or use as default)
                    def.FileName = window.location.pathname.split('/').pop() || urlName;
                }
                def.OutputPath = saveFileAs(def.FileName, content); // Attempt save
            });
        }
    });

    modulo._DEVLIB_SOURCE = (`
    <Artifact name="css" bundle="link[rel=stylesheet]">
        {% for elem in bundle %}{{ elem.bundledContent|default:''|safe }}
        {% endfor %}{% for css in assets.cssAssetsArray %}
        {{ css|safe }}{% endfor %}
```

```
</Artifact>
<Artifact name="js" bundle="script[src]">
   window.moduloBuild = window.moduloBuild || { modules: {}, nameToHash: {} };
   {% for name, hash in assets.nameToHash %}{% if hash in assets.moduleSources %}{% if
name|first is not "_" %}
      window.moduloBuild.modules["{{ hash }}"] = function {{ name }} (modulo) {
         {{ assets.moduleSources|get:hash|safe }}
      };
      window.moduloBuild.nameToHash.{{ name }} = "{{ hash }}";
   {% endif %}{% endif %}{% endfor %}
   window.moduloBuild.definitions = { {% for name, value in definitions %}
      {% if name|first is not "_" %}{{ name }}: {{ value|json|safe }},{% endif %}
   {% endfor %} };
   {% if bundle %}
      {% for elem in bundle %}{{ elem.bundledContent|default:''|safe }}{% endfor %}
      modulo.assets.modules = window.moduloBuild.modules;
      modulo.assets.nameToHash = window.moduloBuild.nameToHash;
      modulo.definitions = window.moduloBuild.definitions;
   {% endif %}
   {% for name in assets.mainRequires %}
      modulo.assets.require("{{ name|escapejs }}");
   {% endfor %}
</Artifact>
<Artifact name="html" url-name="index.html"
remove="script[src],link[href],[modulo-asset],template[modulo],script[modulo],modulo">
   {{ htmlPrefix|safe }}<link rel="stylesheet" href="{{ definitions._artifact_css.OutputPath }}" />
   {{ htmlInterfix|safe }}<script src="{{ definitions._artifact_js.OutputPath }}"></s` + `cript>
   {{ htmlSuffix|safe }}
</Artifact>
`).replace(/^\s+/gm, ''); // Removing indentation from DEVLIB_SOURCE

modulo.config.component = {
   tagAliases: { }, //tagAliases: { 'html-table': 'table', 'html-script': 'script' }, // shortcut for syntax
issues
   mode: 'regular',
   rerender: 'event',
   engine: 'Reconciler', // TODO: RM, dead code
   Contains: 'cparts',
   CustomElement: 'window.HTMLElement',
   DefinedAs: 'name',
   BuildLifecycle: 'build',
   RenderObj: 'component', // Make features available as "renderObj.component"
   // Children: 'cparts', // How we can implement Parentage: Object.keys((get('modulo.registry.' +
value))// cparts))
```

```
    DefLoaders: [ 'DefTarget', 'DefinedAs', 'Src', 'Content' ],
    DefBuilders: [ 'CustomElement', 'alias|AliasNamespace', 'CodeTemplate' ],
    DefFinalizers: [ 'MainRequire' ],
    BuildCommandBuilders: [ 'Prebuild|BuildLifecycle', 'BuildLifecycle' ],
    //Directives: [ 'slotLoad', 'eventMount', 'eventUnmount', 'dataPropMount', 'dataPropUnmount'
],
    Directives: [ 'eventMount', 'eventUnmount', 'dataPropMount', 'dataPropUnmount' ],
    CodeTemplate: `
        const def = modulo.definitions['{{ def.DefinitionName }}'];
        class {{ def.className }} extends {{ def.baseClass|default:'window.HTMLElement' }} {
            constructor() { super(); this.init(); }
        }
        modulo.registry.utils.initComponentClass(modulo, def, {{ def.className }});
        window.customElements.define(def.TagName, {{ def.className }});
        return {{ def.className }};
    `.replace(/^\s+/gm, ''),
};

modulo.register('coreDef', class Component {
    static CustomElement (modulo, def, value) {
        if (!def.ChildrenNames || def.ChildrenNames.length === 0) {
            console.warn('MODULO: Empty ChildrenNames:', def.DefinitionName);
            return;
        } else if (def.namespace === null || def.alias) { // Auto-gen
            def.namespace = def.namespace || def.DefinitionName;
        } else if (!def.namespace) { // Otherwise default to the Modulo def conf
            def.namespace = modulo.config.namespace || 'x'; // or simply 'x-'
        }
        def.name = def.name || def.DefName || def.Name;
        def.TagName = `${ def.namespace }-${ def.name }`.toLowerCase();
        def.MainRequire = def.DefinitionName;
        def.className =  def.className || `${ def.namespace }_${ def.name }`;
    }

    static BuildLifecycle (modulo, def, value) {
        for (const elem of document.querySelectorAll(def.TagName)) {
            elem.cparts.component._lifecycle([ value ]); // Run the lifecycle
        }
        return true;
    }

    static AliasNamespace (modulo, def, value) {
        const fullAlias = `${ value }-${ def.name }`; // Combine new NS and name
        modulo.config.component.tagAliases[fullAlias] = def.TagName;
    }

    rerender(original = null) {
        if (original) {
            if (this.element.originalHTML === null) {
                this.element.originalHTML = original.innerHTML;
            }
            this.element.originalChildren = Array.from(
                original.hasChildNodes() ? original.childNodes : []);
        }
        this._lifecycle([ 'prepare', 'render', 'dom', 'reconcile', 'update' ]);
    }

    buildCallback() {
        const PRE = 'modulo-mount-'; // Prefix used for attributes
        if (this.element.originalHTML !== this.element.innerHTML) {
            this.element.setAttribute(PRE + 'html', this.element.originalHTML);
        }
        const nodes = Array.from(this.element.querySelectorAll('*'));
        for (const [ node, method, arg ] of this._mountPatchset || []) {
            const { rawName, el } = arg || {}; // Extract needed directive info
            const count = el ? nodes.filter(e => e.contains(el)).length : 0;
            if (count) { // count = number of steps in tree hierarchy (or 0)
                const existing = el.getAttribute(PRE + 'patches') || '';
                if (!existing.includes(count + ',' + rawName)) { // Not a dupe
                    const value = existing + '\n' + count + ',' + rawName;
                    el.setAttribute(PRE + 'patches', value.trim());
                }
            }
        }
    }

    getCurrentRenderObj() {
        return (this.element.eventRenderObj || this.element.renderObj ||
this.element.initRenderObj);
    }

    _lifecycle(lifecycleNames, rObj={}) {
        const renderObj = Object.assign({}, rObj, this.getCurrentRenderObj());
        this.element.renderObj = renderObj;
        this.modulo.lifecycle(this.element.cparts, renderObj, lifecycleNames);
        //this.element.renderObj = null; // ?rendering is over, set to null
    }
```

```
initializedCallback() {
    const { newNode } = this.modulo.registry.utils;
    const opts = { directiveShortcuts: [], directives: [] }; // TODO: Move this to reconciler
    for (const cPart of Object.values(this.element.cparts)) {
        const def = (cPart.def || cPart.conf);
        for (const method of def.Directives || []) {
            const dirName = (def.RenderObj || def.Name) + '.' + method;
            opts.directives[dirName] = cPart;
        }
    }
    this.reconciler = new this.modulo.registry.core.Reconciler(this.modulo, opts);
    this.resolver = new this.modulo.registry.core.ValueResolver(this.modulo);
    const html = this.element.getAttribute('modulo-mount-html'); // Hydrate?
    this._mountRival = html === null ? this.element : newNode(html);
    this.element.originalHTML = html === null ? this.element.innerHTML : html;
}

mountCallback() { // Prepare the element, "hydrating" the "mount-patches"
    const ATTR = 'modulo-mount-patches'; // Attribute used
    const { get } = this.modulo.registry.utils;
    for (const elem of this.element.querySelectorAll(`[${ ATTR }]`)) {
        for (const line of elem.getAttribute(ATTR).split('\n')) {
            const [ count, rawName ] = line.split(','); // Comma seperated
            const nodePath = '.parentNode'.repeat(count).substr(1);
            if (this.element === get(elem, nodePath)) { // It's me!
                this.reconciler.patchDirectives(elem, rawName, 'Mount');
                const newVal = elem.getAttribute(ATTR).replace(line, '');
                elem.setAttribute(ATTR, newVal); // "Consume" line from attr
            }
        }
    }
}

mountRenderCallback() { // First "mount", trigger render & hydration
    this.reconciler.applyPatches(this.reconciler.patches); // From "mount"
    this.rerender(this._mountRival); // render + mount childNodes
    delete this._mountRival; // Clear the temporary reference
    this.element.isMounted = true; // Mark as mounted
}

prepareCallback() {
    return { // Create the initial Component renderObj obj
        originalHTML: this.element.originalHTML, // HTML received at mount
        id: this.id, // Universally unique ID number
```

```
            innerHTML: null, // String to copy (default: null is "no-op")
            innerDOM: null, // Node to copy (default: null sets innerHTML)
            patches: null, // Patch array (default: reconcile vs innerDOM)
            slots: { }, // Populate with slots to be filled when reconciling
        };
    }

    domCallback(renderObj) {
        const { clone, newNode } = this.modulo.registry.utils;
        let { slots, root, innerHTML, innerDOM } = renderObj.component;
        if (this.attrs.mode === 'regular' || this.attrs.mode === 'vanish') {
            root = this.element; // default, use element as root
        } else if (this.attrs.mode === 'shadow') {
            if (!this.element.shadowRoot) {
                this.element.attachShadow({ mode: 'open' });
            }
            root = this.element.shadowRoot; // render into attached shadow
        } else if (this.attrs.mode === 'vanish-into-document') {
            root = this.element.ownerDocument.body; // render into body
        } else if (!root) {
            this.modulo.assert(this.attrs.mode === 'custom-root', 'Bad mode')
        }
        if (innerHTML !== null && !innerDOM) { // Use component.innerHTML as DOM
            innerDOM = newNode(innerHTML);
        }
        if (innerDOM && this.attrs.mode === 'vanish-into-document') {
            for (const el of innerDOM.querySelectorAll('script,link,title,meta')) {
                this.element.ownerDocument.head.append(clone(el)); // Move clone
                el.remove(); // Old one is removed from previous location
            }
        }
        if (innerDOM && this.attrs.mode !== 'shadow') {
            for (const elem of this.element.originalChildren) {
                const name = (elem.getAttribute && elem.getAttribute('slot')) || '';
                elem.remove(); // Remove from DOM so it can't self-match
                if (!(name in slots)) {
                    slots[name] = [ elem ]; // Sorting into new slot arrays
                } else {
                    slots[name].push(elem); // Or pushing into existing
                }
            }
        }
        return { root, innerHTML, innerDOM, slots };
    }
    reconcileCallback(renderObj) {
        let { innerHTML, innerDOM, patches, root, slots } = renderObj.component;
        if (innerDOM) {
            this.reconciler.patches = []; // Reset reconciler patches
            this.reconciler.reconcileChildren(root, innerDOM, slots);
            patches = this.reconciler.patches;
        }
        return { patches, innerHTML }; // TODO remove innerHTML from here
    }

    updateCallback(renderObj) {
        const { patches } = renderObj.component;
        if (patches) {
            this._mountPatchset = this._mountPatchset || patches; // 1st render
            this.reconciler.applyPatches(patches); // Apply patches to DOM
        }
        if (this.attrs.mode === 'vanish') { // If we need to vanish, do it now
            this.element.replaceWith(...this.element.childNodes);
        }
    }

    handleEvent(func, payload, ev) {
        this._lifecycle([ 'event' ]);
        func(payload === undefined ? ev : payload);
        this._lifecycle([ 'eventCleanup' ]);
        if (this.attrs.rerender !== 'manual') { // TODO: Change patch('rerender') to
"requestRerender" (or update the HTMLElement method)
            this.element.rerender(); // always rerender after events
        }
    }

    eventMount({ el, value, attrName, rawName }) {
        const { resolveDataProp } = this.modulo.registry.utils;
        const get = (key, key2) => resolveDataProp(key, el, key2 && get(key2));
        this.modulo.assert(get(attrName), `Not found: ${ rawName }=${ value }`);
        el.moduloEvents = el.moduloEvents || {}; // Attach if not already
        const listen = ev => { // Define a listen event func to run handleEvent
            ev.preventDefault();
            const payload = get(attrName + '.payload', 'payload');
            this.handleEvent(resolveDataProp(attrName, el), payload, ev);
        };
        el.moduloEvents[attrName] = listen;
        el.addEventListener(attrName, listen);
    }

    eventUnmount({ el, attrName }) {
        if (el.moduloEvents) { // TODO: Remove this check
            el.removeEventListener(attrName, el.moduloEvents[attrName]);
            delete el.moduloEvents[attrName];
        }
    }

    dataPropMount({ el, value, attrName, rawName }) { // element,
        // Resolve the given value and attach to dataProps
        if (!el.dataProps) {
            el.dataProps = {};
            el.dataPropsAttributeNames = {};
        }
        const resolver = new this.modulo.registry.core.ValueResolver(// OLD TODO: Global
modulo
                      this.element && this.element.getCurrentRenderObj());
        resolver.set(el.dataProps, attrName + ':', value, true);
        el.dataPropsAttributeNames[rawName] = attrName;
    }

    dataPropUnmount({ el, attrName, rawName }) {
        if (!el.dataProps) { console.log('Modulo ERROR: Could not Unmount', attrName, rawName,
el); }
        if (el.dataProps) {
            delete el.dataProps[attrName];
            delete el.dataPropsAttributeNames[rawName];
        }
    }
});

modulo.register('coreDef', class Library { }, {
    Contains: 'coreDefs',
    DefTarget: 'config.component',
    DefLoaders: [ 'DefTarget', 'DefinedAs', 'Src', 'Content' ],
});

modulo.register('util', function keyFilter (obj, func) {
    const keys = func.call ? Object.keys(obj).filter(func) : func;
    return Object.fromEntries(keys.map(key => [ key, obj[key] ]));
});

modulo.register('util', function resolveDataProp (key, elem, defaultVal) {
    if (elem.dataProps && key in elem.dataProps) {
        return elem.dataProps[key];
    }
    return elem.hasAttribute(key) ? elem.getAttribute(key) : defaultVal;
});

modulo.register('util', function cleanWord (text) {
    // todo: should merge with stripWord ? See if "strip" functionality is enough
    return (text + '').replace(/[^a-zA-Z0-9$_\.]/g, '') || '';
});

modulo.register('util', function stripWord (text) {
    return text.replace(/^[^a-zA-Z0-9$_\.]/, '')
            .replace(/[^a-zA-Z0-9$_\.]$/, '');
});

modulo.register('util', function clone (el) { // Clones an element
    const newElement = window.document.createElement(el.tagName);
    for (const attr of el.attributes) { // Copy all attributes from old elem
        newElement.setAttributeNode(attr.cloneNode(true)); // ...to new elem
    }
    newElement.textContent = el.textContent; // (e.g. <title>Hi</title>)
    if (el.tagName === 'SCRIPT') { // Need to fix SCRIPT tag fluke
        newElement.src = el.src; // (TODO: Still needed?)
    }
    return newElement;
});

modulo.register('util', function hash (str) {
    // Simple, insecure, "hashCode()" implementation. Returns base32 hash
    let h = 0;
    for (let i = 0; i < str.length; i++) {
        //h = ((h << 5 - h) + str.charCodeAt(i)) | 0;
        h = Math.imul(31, h) + str.charCodeAt(i) | 0;
    }
    const hash8 = ('---------' + (h || 0).toString(32)).slice(-8);
    return hash8.replace(/-/g, 'x'); // Pad with 'x'
});

modulo.register('util', function newNode(innerHTML, tag) {
    const obj = { innerHTML }; // Extra properties to assign
    return Object.assign(window.document.createElement(tag || 'div'), obj);
});
modulo.registry.utils.makeDiv = modulo.registry.utils.newNode; // TODO: Rm alias
modulo.register('util', function normalize(html) {
    // Normalize space to ' ' & trim around tags
    return html.replace(/\s+/g, ' ').replace(/(^|>)\s*(<|$)/g, '$1$2').trim();
});

modulo.register('util', function escapeRegExp(s) {
    return s.replace(/[.*+?^${}()|[\]\\]/g, "\\" + "\x24" + "&");
});

modulo.register('util', function saveFileAs(filename, text) {
    const element = window.document.createElement('a');
    const enc = window.encodeURIComponent(text);
    element.setAttribute('href', 'data:text/plain;charset=utf-8,' + enc);
    element.setAttribute('download', filename);
    window.document.body.appendChild(element);
    element.click();
    window.document.body.removeChild(element);
    return `./${filename}`; // by default, return local path
});

modulo.register('util', function get(obj, key) {
    return (key in obj) ? obj[key] : (key + '').split('.').reduce((o, name) => o[name], obj);
});

modulo.register('util', function set(obj, keyPath, val) {
    return new window.modulo.registry.core.ValueResolver(window.modulo).set(obj, keyPath,
val);
});

modulo.register('util', function getParentDefPath(modulo, def) {
    const { getParentDefPath } = modulo.registry.utils; // Use to recurse
    const pDef = def.Parent ? modulo.definitions[def.Parent] : null;
    const url = String(window.location).split('?')[0]; // Remove ? and #
    return pDef ? pDef.Source || getParentDefPath(modulo, pDef) : url;
});

modulo.register('core', class AssetManager {
    constructor (modulo) { // TODO: Refactor this class into util / processors
        this.modulo = modulo;
        this.stylesheets = {}; // Object with hash of CSS (prevents double add)
        this.cssAssetsArray = []; // List of CSS assets added, in order
        this.modules = {}; // Object containing JS functions with hashed keys
        this.moduleSources = {}; // Source code of JS functions (for build)
```

```
        this.nameToHash = {}; // Reversable hash / human name for modules
        this.mainRequires = []; // List of globally invoked modules
    }

    mainRequire(moduleName) {
        this.mainRequires.push(moduleName);
        this.require(moduleName);
    }

    require(moduleName) {
        // TODO: Don't use nameToHash for simpler look-up, but include
        // "hashToName" for deduping during add (just create extra refs)
        this.modulo.assert(moduleName in this.nameToHash,
            `${ moduleName } not in ${ Object.keys(this.nameToHash).join(', ') }`);
        const hash = this.nameToHash[moduleName];
        this.modulo.assert(hash in this.modules,
            `${ moduleName } / ${ hash } not in ${ Object.keys(this.modules).join(', ') }`);
        return this.modules[hash].call(window, this.modulo);
    }

    define(name, code) {
        const hash = this.modulo.registry.utils.hash(code);
        this.modulo.assert(!(name in this.nameToHash), `Duplicate: ${ name }`);
        this.nameToHash[name] = hash;
        if (!(hash in this.modules)) {
            this.moduleSources[hash] = code;
            const assignee = `window.modulo.assets.modules["${ hash }"] = `;
            const prefix = assignee + `function ${ name } (modulo) {\n`;
            this.appendToHead('script', `"use strict";${ prefix }${ code }};\n`);
        }
    }

    appendToHead(tagName, codeStr) {
        const doc = window.document;
        if (!doc || !doc.head) { // Fall back to just using a Function
            return new Function('window', 'modulo', codeStr)(window, this.modulo);
        }
        const elem = doc.createElement(tagName);
        elem.setAttribute('modulo-asset', 'y'); // Mark as an "asset"
        doc.head.append(elem);
        elem.textContent = codeStr; // Blocking, causes eval
    }
});
```

```
modulo.register('core', class FetchQueue {
    constructor(modulo, queue = {}, data = {}) {
        Object.assign(this, { modulo, queue, data });
        this.wait = callback => this.enqueue(callback, true); // TODO: RM this alias
    }

    fetch(src) {  // Returns "thennable" that somewhat resembles window.fetch
        return { then: callback => this.request(src, callback, console.error) };
    }

    request(src, resolve, reject) { // Do fetch & do enqueue
        if (src in this.data) { // Cached data found
            resolve(this.data[src], src); // (sync route)
        } else if (!(src in this.queue)) { // No cache, no queue
            this.queue[src] = [ resolve ]; // First time, create the queue Array
            const { force, callbackName } = this.modulo.config.fetchqueue;
            if ((!force && src.startsWith('file:/')) || force === 'file') {
                window[callbackName] = str => { this.__data = str };
                const elem = window.document.createElement('SCRIPT');
                elem.onload = () => this.receiveData(this.__data, src);
                elem.src = src + (src.endsWith('/') ? 'index.html' : '');
                elem.setAttribute('modulo-asset', 'y'); // Stay out of build
                window.document.head.append(elem); // Actually execute request
            } else { // Otherwise, use normal fetch transport method
                window.fetch(src, { cache: 'no-store' })
                    .then(response => response.text())
                    .then(text => this.receiveData(text, src))
                    .catch(reject);
            }
        } else { // Otherwise: Already requested, only enqueue function
            this.queue[src].push(resolve);
        }
    }

    receiveData(text, src) { // Receive data, optionally trimming padding
        const { prefix, suffix } = this.modulo.config.fetchqueue.filePadding;
        if (text && text.startsWith(prefix) && prefix && text.trim().endsWith(suffix)) {
            text = text.trim().slice(prefix.length, 0 - suffix.length); // Clean
        }
        this.data[src] = text; // Keep retrieved data cached here for sync route
        const resolveCallbacks = this.queue[src];
        delete this.queue[src];
        for (const dataCallback of resolveCallbacks) { // Loop through callbacks
            dataCallback(text, src);
        }
    }

    enqueue(callback, waitForAll = false) { // Wait for _current_ queue (or all)
        const allQueues = Array.from(Object.values(this.queue)); // Copy array
        if (allQueues.length === 0) {
            return callback();
        } else if (waitForAll) { // Doing a "wait()" -- need to re-enqueue
            return this.enqueue(() => Object.keys(this.queue).length === 0 ?
                                callback() : this.enqueue(callback, true));
        }
        let count = 0; // Using count we only do callback() when ALL returned
        const check = () => ((++count >= allQueues.length) ? callback() : 0);
        allQueues.forEach(queue => queue.push(check)); // Add to every queue
    }
}, {
    callbackName: 'DOCTYPE_MODULO',
    filePadding: { prefix: '!DOCTYPE_MODULO(`', suffix: '`)' },
});

modulo.register('cpart', class Props {
    initializedCallback(renderObj) {
        const props = {};
        const { resolveDataProp } = this.modulo.registry.utils;
        for (const [ propName, def ] of Object.entries(this.attrs)) {
            props[propName] = resolveDataProp(propName, this.element, def);
            // TODO: Implement type-checked, and required
        }
        return props;
    }

    prepareCallback(renderObj) {
        /* TODO: Remove after observedAttributes is implemented, e.g.:
          static factoryCallback({ attrs }, { componentClass }, renderObj) {
              //componentClass.observedAttributes = Object.keys(attrs);
          }
        */
        return this.initializedCallback(renderObj);
    }
});

modulo.config.style = {
    AutoIsolate: true, // true is "default behavior" (autodetect)
    urlReplace: null, // null is "default behavior" (only if -src is specified)
    isolateSelector: null, // Later has list of selectors
    isolateClass: null, // By default, it does not use class isolation
    prefix: null, // Used to specify prefix-based isolation (most common)
    corePseudo: ['before', 'after', 'first-line', 'last-line' ],
    DefBuilders: [ 'FilterContent', 'AutoIsolate', 'Content|ProcessCSS' ],
};
modulo.register('cpart', class Style {
    static AutoIsolate(modulo, def, value) {
        const { AutoIsolate } = modulo.registry.cparts.Style; // (for recursion)
        const { namespace, mode, Name } = modulo.definitions[def.Parent] || {};
        if (value === true) { // Auto-detect based on parent's mode
            AutoIsolate(modulo, def, mode); // Check "mode" instead (1x recurse)
        } else if (value === 'regular' && !def.isolateClass) { // Use prefix
            def.prefix = def.prefix || `${ namespace }-${ Name }`;
        } else if (value === 'vanish') { // Vanish-based, specify "isolateClass"
            def.isolateClass = def.isolateClass || def.Parent;
        }
    }
    static processSelector (modulo, def, selector) {
        const hostPrefix = def.prefix || ('.' + def.isolateClass);
        if (def.isolateClass || def.prefix) {
            // Upgrade the ":host" or :root pseudo-elements to be the full name
            const hostRegExp = new RegExp(/:(host|root)(\([^)]*\))?/, 'g');
            selector = selector.replace(hostRegExp, hostClause => {
                hostClause = hostClause.replace(/:(host|root)/gi, '');
                return hostPrefix + (hostClause ? `:is(${ hostClause })` : '');
            });
        }
        let selectorOnly = selector.replace(/\s*[\{,]\s*,?$/, '').trim();
        if (def.isolateClass && selectorOnly !== hostPrefix) {
            // Remove extraneous characters (and strip ',' for isolateSelector)
            let suffix = /{\s*$/.test(selector) ? ' {' : ', ';
            selectorOnly = selectorOnly.replace(/:(:?[a-z-]+)\s*$/i, (all, pseudo) => {
                if (pseudo.startsWith(':') || def.corePseudo.includes(pseudo)) {
                    suffix = ':' + pseudo + suffix; // Attach to suffix, on outside
                    return ''; // Strip pseudo from the selectorOnly variable
                }
                return all;
            });
            def.isolateSelector.push(selectorOnly); // Add to array for later
            selector = `.${ def.isolateClass }:is(${ selectorOnly })` + suffix;
        }
        if (def.prefix && !selector.startsWith(def.prefix)) {
            // A prefix was specified, so prepend it if it doesn't have it
            selector = `${ def.prefix } ${ selector }`;
        }
        return selector;
    }
    static ProcessCSS (modulo, def, value) {
        if (def.isolateClass || def.prefix) {
            if (!def.keepComments) {
                value = value.replace(/\/\*.+?\*\//g, ''); // strip comments
            }
            def.isolateSelector = []; // Used to accumulate elements to select
            value = value.replace(/([^\r\n,{}]+)(,(?=[^}]*{)|\s*{)/gi, selector => {
                selector = selector.trim();
                if (selector.startsWith('@') || selector.startsWith('from')
                                    || selector.startsWith('to')) {
                    return selector; // Skip (e.g. is @media or @keyframes)
                }
                return this.processSelector(modulo, def, selector);
            });
        }
        if (def.urlReplace || (def.urlReplace === null && def.Source)) {
            const key = def.urlReplace === 'absolute' ? 'href' : 'pathname';
            value = value.replace(/url\(['"]?([^)]+?)['"]?\)/gi, (all, url) => {
                if (url.startsWith('.')) { // If relative, make absolute
                    return `url("${ (new window.URL(url, def.Source))[key] }")`;
                }
                return all; // Not a relative URL, return all text untampered
            });
        }
        const { mode } = modulo.definitions[def.Parent] || {};
        if (mode === 'shadow') { // Stash in the definition configuration
            def.shadowContent = (def.shadowContent || '') + value;
        } else { // Otherwise, just "register" as a modulo asset
            def.headHash = modulo.registry.utils.hash(value);
            if (!(def.headHash in modulo.assets.stylesheets)) {
                modulo.assets.stylesheets[def.headHash] = value;
                modulo.assets.cssAssetsArray.push(value);
            }
        }
    }
    makeStyleTag(parentElem, content) { // Append <style> tag with content
        const style = window.document.createElement('style');
        style.setAttribute('modulo-asset', 'y');
        style.textContent = content;
        parentElem.append(style);
    }
    domCallback(renderObj) {
        const { mode } = modulo.definitions[this.conf.Parent] || {};
        const { innerDOM, Parent } = renderObj.component;
        const { headHash, isolateClass, isolateSelector, shadowContent } = this.conf;
        if (isolateClass && isolateSelector && innerDOM) { // Attach classes
            const selector = isolateSelector.filter(s => s).join(',\n');
            for (const elem of innerDOM.querySelectorAll(selector)){
                elem.classList.add(isolateClass);
            }
        }
        if (shadowContent && innerDOM) { // Append to element to reconcile
            this.makeStyleTag(innerDOM, shadowContent);
        }
        const devStyles = this.modulo.assets.stylesheets;
        if (headHash && headHash in devStyles) { // Add to page: Hasn't been yet
            this.makeStyleTag(window.document.head, devStyles[headHash]);
            delete this.conf.headHash; // Consume, so it only happens once
        }
    }
});

modulo.register('cpart', class Template {
    static TemplatePrebuild (modulo, def, value) {
        modulo.assert(def.Content, `Empty Template: ${def.DefinitionName}`);
        const template = modulo.instance(def, { });
        // def.Content = def.Content.trim(); // XXX TODO BREAKS IA ALL?
        const compiledCode = template.compileFunc(def.Content);
        const code = `return function (CTX, G) { ${ compiledCode } };`;
        // TODO: Refactor this to use define processor?
        modulo.assets.define(def.DefinitionName, code);
        delete def.Content;
    }
    constructor(text, options = null) {
        if (typeof text === 'string') { // Using "new" (direct JS interface)
            window.modulo.instance({ }, options || { }, this); // Setup object
            this.conf.DefinitionName = '_template_template' + this.id; // Unique
            const code = `return function (CTX, G) { ${ this.compileFunc(text) } };`;
            this.modulo.assets.define(this.conf.DefinitionName, code);
        }
    }

    constructedCallback() {
        this.stack = []; // Parsing tag stack, used to detect unclosed tags
        // Combine conf from all sources: config, defaults, and "registered"
        const { filters, tags } = this.conf;
        const { defaultFilters, defaultTags } = this.modulo.config.template;
        const { templateFilters, templateTags } = this.modulo.registry;
        Object.assign(this, this.modulo.config.template, this.conf);
        // Set "filters" and "tags" with combined / squashed configuration
        this.filters = Object.assign({}, defaultFilters, templateFilters, filters);
        this.tags = Object.assign({}, defaultTags, templateTags, tags);
    }

    initializedCallback() {
        return { render: this.render.bind(this) }; // Export "render" method
    }

    renderCallback(renderObj) {
        if (this.conf.Name === 'template' || this.conf.active) { // If primary
            renderObj.component.innerHTML = this.render(renderObj); // Do render
        }
    }

    parseExpr(text) {
        // Output JS code that evaluates an equivalent template code expression
        const filters = text.split('|');
        let results = this.parseVal(filters.shift()); // Get left-most val
        for (const [ fName, arg ] of filters.map(s => s.trim().split(':'))) {
            // TODO: Store a list of variables / paths, so there can be
            // warnings or errors when variables are unspecified
            const argList = arg ? ',' + this.parseVal(arg) : '';
            results = `G.filters["${fName}"](${results}${argList})`;
        }
        return results;
    }

    parseCondExpr(string) {
        // Return an Array that splits around ops in an "if"-style statement
        const regExpText = ` (${this.opTokens.split(',').join('|')}) `;
        return string.split(RegExp(regExpText));
    }

    toCamel(string) { // Takes kebab-case and converts toCamelCase
        return string.replace(/-([a-z])/g, g => g[1].toUpperCase());
    }

    parseVal(string) {
        // Parses str literals, de-escaping as needed, numbers, and context vars
        const { cleanWord } = modulo.registry.utils; // TODO: RM this "safety"
        const s = string.trim();
        if (s.match(/^('.*'|".*")$/)) { // String literal
            return JSON.stringify(s.substr(1, s.length - 2));
        }
        return s.match(/^\d+$/) ? s : `CTX.${ cleanWord(this.toCamel(s)) }`
    }

    escapeText(text) {
        if (text && text.safe) {
            return text;
        }
        return (text + '').replace(/&/g, '&amp;')
            .replace(/</g, '&lt;').replace(/>/g, '&gt;')
            .replace(/'/g, '&#x27;').replace(/"/g, '&quot;');
    }

    tokenizeText(text) {
        // Join all modeTokens with | (OR in regex)
        const { escapeRegExp } = this.modulo.registry.utils;
        const re = '(' + this.modeTokens.map(escapeRegExp).join('|(').replace(/ +/g, ')(.+?)');
        return text.split(RegExp(re)).filter(token => token !== undefined);
    }

    compileFunc(text) {
        const { normalize } = this.modulo.registry.utils;
        let code = 'var OUT=[];\n'; // Variable used to accumulate code
        let mode = 'text'; // Start in text mode
        const tokens = this.tokenizeText(text);
        for (const token of tokens) {
            if (mode) { // If in a "mode" (text or token), then call mode func
                const result = this.modes[mode](token, this, this.stack);
                if (result) { // Mode generated text output, add to code
                    const comment = !this.disableComments ? '' :
                        ' // ' + JSON.stringify(normalize(token).trim());
                    code += `  ${ result }${ comment }\n`;
                }
            }
            // FSM for mode: ('text' -> null) (null -> token) (* -> 'text')
            mode = (mode === 'text') ? null : (mode ? 'text' : token);
        }
        code += '\nreturn OUT.join("");'
        const unclosed = this.stack.map(({ close }) => close).join(', ');
```

```
            this.modulo.assert(!unclosed, `Unclosed tags: ${ unclosed }`);
            return code;
        }

        render(renderObj) {
            if (!this.renderFunc) { // Run module and get function
                this.renderFunc = this.modulo.assets.require(this.conf.DefinitionName);
            }
            return this.renderFunc(Object.assign({ renderObj }, renderObj), this);
        }
    }, {
        TemplatePrebuild: "y", // TODO: Refactor
        DefFinalizers: [ 'FilterContent', 'TemplatePrebuild' ],
        FilterContent: 'trim|tagswap:config.component.tagAliases',
        // textFilter: 'escape|etc', // TODO
        // TODO: Consider reserving "x" and "y" as temp vars, e.g.  // (x = X, y = Y).includes ?
y.includes(x) : (x in y)
        opTokens: '==,>,<,>=,<=,!=,not in,is not,is,in,not,gt,lt',
        opAliases: {
            '==': 'X === Y',
            'is': 'X === Y',
            'gt': 'X > Y',
            'lt': 'X < Y',
            'is not': 'X !== Y',
            'not': '!(Y)',
            'in': '(Y).includes ? (Y).includes(X) : (X in Y)',
            'not in': '!((Y).includes ? (Y).includes(X) : (X in Y))',
        },
    });

    modulo.config.template.modeTokens = [ '{% %}', '{{ }}', '{# #}' ];
    modulo.config.template.modes = {
        '{%': (text, tmplt, stack) => {
            const tTag = text.trim().split(' ')[0];
            const tagFunc = tmplt.tags[tTag];
            if (stack.length && tTag === stack[stack.length - 1].close) {
                return stack.pop().end; // Closing tag, return it's end code
            } else if (!tagFunc) { // Undefined template tag
                throw new Error(`Unknown template tag "${tTag}": ${text}`);
            } // Normal opening tag
            const result = tagFunc(text.slice(tTag.length + 1), tmplt);
            if (result.end) { // Not self-closing, push to stack
                stack.push({ close: `end${ tTag }`, ...result });
            }
            return result.start || result;
        },
        '{#': (text, tmplt) => false, // falsy values are ignored
        '{{': (text, tmplt) => `OUT.push(G.escapeText(${tmplt.parseExpr(text)}));`,
        text: (text, tmplt) => text && `OUT.push(${JSON.stringify(text)});`,
    };

    // TODO: Tidy up: Move most or all definitions loose like modes
    modulo.config.template.defaultFilters = (function () {
        const { get } = modulo.registry.utils;
        const safe = s => Object.assign(new String(s), { safe: true });
        const escapeForRE = s=> s.replace(/[.*+?^${}()|[\]\\]/g, '\\$&');
        const trim = (s, arg) => { // General purpose string trimming function
            arg = arg ? escapeForRE(arg).replace(',', '|') : '|';
            return s.replace(new RegExp(`^\\s*${ arg }\\s*$`, 'g'), '');
        };
        const tagswap = (s, arg) => {
            arg = typeof arg === 'string' ? arg.split(/\s+/) : Object.entries(arg);
            for (const row of arg) { // Loop through each replacement pair
                const [ tag, val ] = typeof row === 'string' ? row.split('=') : row;
                const swap = (a, prefix, suffix) => prefix + val + suffix;
                //s = s.replace(RegExp('(</?)' + tag + '(\s|>)', 'gi'),  swap);
                s = s.replace(RegExp('(</?)' + tag + '(\\s|>)', 'gi'),  swap);
            }
            return safe(s); // Always mark as safe, since for HTML tags
        };
        const filters = {
            add: (s, arg) => s + arg,
            allow: (s, arg) => arg.split(',').includes(s) ? s : '',
            camelcase: s => s.replace(/-([a-z])/g, g => g[1].toUpperCase()),
            capfirst: s => s.charAt(0).toUpperCase() + s.slice(1),
            combine: (s, arg) => s.concat ? s.concat(arg) : Object.assign({}, s, arg),
            default: (s, arg) => s || arg,
            divisibleby: (s, arg) => ((s * 1) % (arg * 1)) === 0,
            dividedinto: (s, arg) => Math.ceil((s * 1) / (arg * 1)),
            escapejs: s => JSON.stringify(String(s)).replace(/(^"|"$)/g, ''),
            first: s => Array.from(s)[0],
            join: (s, arg) => (s || []).join(arg === undefined ? ", " : arg),
            json: (s, arg) => JSON.stringify(s, null, arg || undefined),
            last: s => s[s.length - 1],
            length: s => s.length !== undefined ? s.length : Object.keys(s).length,
            lower: s => s.toLowerCase(),
            multiply: (s, arg) => (s * 1) * (arg * 1),
            number: (s) => Number(s),
            pluralize: (s, arg) => (arg.split(',')[(s === 1) * 1]) || '',
            skipfirst: (s, arg) => Array.from(s).slice(arg || 1),
            subtract: (s, arg) => s - arg,
            truncate: (s, arg) => ((s && s.length > arg*1) ? (s.substr(0, arg-1) + '…') : s),
            type: s => s === null ? 'null' : (Array.isArray(s) ? 'array' : typeof s),
            renderas: (rCtx, template) => safe(template.render(rCtx)),
            reversed: s => Array.from(s).reverse(),
            upper: s => s.toUpperCase(),
            yesno: (s, arg) => `${ arg || 'yes,no' },,`.split(',')[s ? 0 : s === null ? 2 : 1],
        };
        const { values, keys, entries } = Object;
        const extra = { trim, tagswap, get, safe, values, keys, entries };
        return Object.assign(filters, extra);
    })();

    modulo.config.template.defaultTags = {
        'debugger': () => 'debugger;',
        'if': (text, tmplt) => {
            // Limit to 3 (L/O/R)
            const [ lHand, op, rHand ] = tmplt.parseCondExpr(text);
            const condStructure = !op ? 'X' : tmplt.opAliases[op] || `X ${op} Y`;
            const condition = condStructure.replace(/([XY])/g,
                (k, m) => tmplt.parseExpr(m === 'X' ? lHand : rHand));
            const start = `if (${condition}) {`;
            return { start, end: '}' };
        },
        'else': () => '} else {',
        'elif': (s, tmplt) => '} else ' + tmplt.tags['if'](s, tmplt).start,
        'comment': () => ({ start: "/*", end: "*/"}),
        'include': (text) => `OUT.push(CTX.${ text.trim() }.render(CTX));`,
        'for': (text, tmplt) => {
            // Make variable name be based on nested-ness of tag stack
            const { cleanWord } = modulo.registry.utils;
            const arrName = 'ARR' + tmplt.stack.length;
            const [ varExp, arrExp ] = text.split(' in ');
            let start = `var ${arrName}=${tmplt.parseExpr(arrExp)};`;
            // TODO: Upgrade to for...of loop (after good testing)
            start += `for (var KEY in ${arrName}) {`;
            const [keyVar, valVar] = varExp.split(',').map(cleanWord);
            if (valVar) {
                start += `CTX.${keyVar}=KEY;`;
            }
            start += `CTX.${valVar ? valVar : varExp}=${arrName}[KEY];`;
            return { start, end: '}'};
        },
        'empty': (text, {stack}) => {
            // Make variable name be based on nested-ness of tag stack
            const varName = 'G.FORLOOP_NOT_EMPTY' + stack.length;
            const oldEndCode = stack.pop().end; // get rid of dangling for
            const start = `${varName}=true; ${oldEndCode} if (!${varName}) {`;
            const end = `}${varName} = false;`;
            return { start, end, close: 'endfor' };
        },
    };

    modulo.register('processor', function contentCSV (modulo, def, value) {
        const parse = s => s.trim().split('\n').map(line => line.trim().split(','));
        def.Code = 'return ' + JSON.stringify(parse(def.Content || ''));
    });

    modulo.register('processor', function contentJS (modulo, def, value) {
        const tmpFunc = new Function('return (' + (def.Content || 'null') + ');');
        def.Code = 'return ' + JSON.stringify(tmpFunc()) + ';'; // Evaluate
    });

    modulo.register('processor', function contentJSON (modulo, def, value) {
        def.Code = 'return ' + JSON.stringify(JSON.parse(def.Content || '{}')) + ';';
    });

    modulo.register('processor', function contentTXT (modulo, def, value) {
        def.Code = 'return ' + JSON.stringify(def.Content);
    });

    modulo.register('processor', function dataType (modulo, def, value) {
        if (value === '?') { // '?' means determine based on extension
            const ext = def.Src && def.Src.match(/(?<=\.)[a-z]+$/i);
            value = ext ? ext[0] : 'json';
        }
        def['Content' + value.toUpperCase()] = value;
    });

    modulo.register('processor', function filterContent (modulo, def, value) {
        if (def.Content && value) { // Check if active (needs truthy value)
            // value = value.trim().replace(/\s*\n\s*\|/gi, '|'); // Would allow multi-line
            const miniTemplate = `{{ def.Content|${ value }|safe }}`;
            const tmplt = new modulo.registry.cparts.Template(miniTemplate);
            def.Content = tmplt.render({ def, config: modulo.config });
        }
```

```
});

modulo.register('processor', function code (modulo, def, value) {
    if (def.DefinitionName in modulo.assets.nameToHash) {
        console.error("ERROR: Duped def:", def.DefinitionName);
        return;
    }
    // TODO: Refactor this inline, change modulo.assets into plain object
    modulo.assets.define(def.DefinitionName, value);
});

modulo.register('processor', function codeTemplate (modulo, def, value) {
    if (def.DefinitionName in modulo.assets.nameToHash) {
        console.error("ERROR: Duped def:", def.DefinitionName);
        return;
    }
    const tmplt = new modulo.registry.cparts.Template(value); // Compile Template
    modulo.assets.define(def.DefinitionName, tmplt.render({ modulo, def }));
});

modulo.register('cpart', class StaticData {
    static RequireData (modulo, def, value) {
        def.data = modulo.assets.require(def[value]); // TODO: Remove asset
    }
    prepareCallback() {
        return this.conf.data;
    }
}, {
    DataType: '?', // Default behavior is to guess based on Src ext
    RequireData: 'DefinitionName',
    DefLoaders: [ 'DefTarget', 'DefinedAs', 'DataType', 'Src', 'FilterContent' ],
    DefBuilders: [ 'ContentCSV', 'ContentTXT', 'ContentJSON', 'ContentJS', 'Code', 'RequireData'
],
});

modulo.register('coreDef', class Configuration { }, {
    DefTarget: 'config',
    DefLoaders: [ 'DefTarget', 'DefinedAs', 'Src|SrcSync', 'Content|Code',
'DefinitionName|MainRequire' ],
});

modulo.config.script = {
    lifecycle: null,
    DefBuilders: [ 'Content|AutoExport', 'CodeTemplate' ],
    CodeTemplate: `var script = { exports: { } };{% if def.locals.length %}
    var {{ def.locals|join:',' }};{% endif %}
    {{ def.tempContent|safe }}
    ;return { exports: script.exports,
        {% for n in def.exportNames %}
            "{{ n }}": typeof {{ n }} !== "undefined" ? {{ n }} : undefined,
        {% endfor %}
        setLocalVariables: function (o) {
            {% for n in def.locals %}{{ n }} = o.{{ n }};{% endfor %}
        }
    }`.replace(/\s\s+/g, ' '),
};

modulo.register('cpart', class Script {
    static AutoExport (modulo, def, value) {
        const { getAutoExportNames } = modulo.registry.utils;
        if (def.lifecycle && def.lifecycle !== 'initialized') {
            value = `function ${ def.lifecycle }Callback (renderObj) {${ value }}`;
        }
        const { ChildrenNames } = modulo.definitions[def.Parent] || { };
        const sibs = (ChildrenNames || []).map(n => modulo.definitions[n].Name);
        sibs.push('component', 'element', 'cparts'); // Add in extras
        def.exportNames = def.exportNames || getAutoExportNames(value); // Scan names
        def.locals = def.locals || sibs.filter(name => value.includes(name));
        def.Directives = def.exportNames.filter(s => s.match(/(Unmount|Mount)$/));
        def.tempContent = value; // TODO: Refactor AutoExport + CodeTemplate
    }

    static factoryCallback(renderObj, def, modulo) {
        //modulo.assert(results || !def.Parent, 'Invalid script return');
        const hash = modulo.assets.nameToHash[def.DefinitionName];
        const func = () => modulo.assets.modules[hash].call(window, modulo);
        if (def.lifecycle === 'initialized') {
            return { initializedCallback: func }; // Attach as callback
        }
        return func(); // Otherwise, should run in static context (e.g. now)
    }

    initializedCallback(renderObj) {
        // Create all lifecycle callbacks, wrapping around the inner script
        const script = renderObj[this.conf.Name];
        this.eventCallback = (rObj) => { // Create eventCallback to set inner
            const vars = { element: this.element, cparts: this.element.cparts };
            const setLocal = script.setLocalVariables || (() => {});
            setLocal(Object.assign(vars, rObj)); // Set inner vars (or no-op)
        };

        if (script.initializedCallback) { // If defined, trigger inner init
            this.eventCallback(renderObj); // Prep before (used by lc=false)
            Object.assign(script, script.initializedCallback(renderObj));
            this.eventCallback(renderObj); // Prep again (used by lc=initialize)
        }

        const isCallback = /(Mount|Unmount|Callback)$/;
        for (const cbName of Object.keys(script)) {
            if (cbName === 'initializedCallback' || !cbName.match(isCallback)) {
                continue; // Skip over initialized (already handled) and non-CBs
            }
            this[cbName] = arg => { // Arg: Either renderObj or directive obj
                const renderObj = this.element.getCurrentRenderObj();
                const script = renderObj[this.conf.Name]; // Get new render obj
                this.eventCallback(renderObj); // Prep before lifecycle method
                Object.assign(script, script[cbName](arg) || {});
            };
        }
    }
});


modulo.register('cpart', class State {
    static factoryCallback(renderObj, def, modulo) {
        if (def.Store) {
            const store = modulo.registry.utils.makeStore(modulo, def);
            if (!(def.Store in modulo.stores)) {
                modulo.stores[def.Store] = store;
            } else {
                Object.assign(modulo.stores[def.Store].data, store.data);
            }
        }
    }

    initializedCallback(renderObj) {
        const store = this.conf.Store ? this.modulo.stores[this.conf.Store]
                : this.modulo.registry.utils.makeStore(this.modulo, this.conf);
        store.subscribers.push(Object.assign(this, store));
        this.types = { range: Number, number: Number, checkbox: (val, el) => el.checked };
        return store.data; // TODO: Possibly, push ALL sibling CParts with stateChangedCallback
    }
```

```
bindMount({ el, attrName, value }) {
    const name = attrName || el.getAttribute('name');
    const val = this.modulo.registry.utils.get(this.data, name);
    this.modulo.assert(val !== undefined, `state.bind "${name}" undefined`);
    const isText = el.tagName === 'TEXTAREA' || el.type === 'text';
    const evName = value ? value : (isText ? 'keyup' : 'change');
    if (!(name in this.boundElements)) {
        this.boundElements[name] = [];
    }
    // Bind the "listen" event to propagate to all, and trigger initial vals
    const listen = () => this.propagate(name, el.value, el);
    this.boundElements[name].push([ el, evName, listen ]);
    el.addEventListener(evName, listen); // todo: make optional, e.g. to support cparts?
    this.propagate(name, val, this); // trigger initial assignment(s)
}

bindUnmount({ el, attrName }) {
    const name = attrName || el.getAttribute('name');
    if (!(name in this.boundElements)) { // TODO HACK
        return console.log('Modulo ERROR: Could not unbind', name);
    }
    const remainingBound = [];
    for (const row of this.boundElements[name]) {
        if (row[0] === el) {
            row[0].removeEventListener(row[1], row[2]);
        } else {
            remainingBound.push(row);
        }
    }
    this.boundElements[name] = remainingBound;
}

stateChangedCallback(name, value, el) {
    this.modulo.registry.utils.set(this.data, name, value);
    if (!this.conf.Only || this.conf.Only.includes(name)) { // TODO: Test
        this.element.rerender();
    }
}

eventCallback() {
    this._oldData = Object.assign({}, this.data);
}
```

```
    propagate(name, val, originalEl = null) {
        const elems = (this.boundElements[name] || []).map(row => row[0]);
        const typeConv = this.types[ originalEl ? originalEl.type : null ];
        val = typeConv ? typeConv(val, originalEl) : val; // Apply conversion
        for (const el of this.subscribers.concat(elems)) {
            if (originalEl && el === originalEl) {
                continue; // don't propagate to originalEl (avoid infinite loop)
            }
            if (el.stateChangedCallback) { // A callback was found, use instead
                el.stateChangedCallback(name, val, originalEl);
            } else if (el.type === 'checkbox') { // Check input use ".checkbox"
                el.checked = !!val;
            } else { // Normal inputs use ".value"
                el.value = val;
            }
        }
    }

    eventCleanupCallback() {
        for (const name of Object.keys(this.data)) {
            this.modulo.assert(!this.conf.AllowNew && name in this._oldData,
                `State variable "${ name }" is undeclared (no "-allow-new")`);
            if (this.data[name] !== this._oldData[name]) {
                this.propagate(name, this.data[name], this);
            }
        }
        this._oldData = null;
    }
}, {
    Directives: [ 'bindMount', 'bindUnmount' ],
    Store: null,
});

modulo.register('utils', class DOMCursor {
    constructor(parentNode, parentRival, slots) {
        this.slots = slots || {}; // Slottables keyed by name (default is '')
        this.instanceStack = []; // Used for implementing DFS non-recursively
        this._rivalQuerySelector = parentRival.querySelector.bind(parentRival);
        this._querySelector = parentNode.querySelector.bind(parentNode);
        this.initialize(parentNode, parentRival);
    }

    initialize(parentNode, parentRival) {
        this.parentNode = parentNode;
        this.nextChild = parentNode.firstChild;
        this.nextRival = parentRival.firstChild;
        this.keyedChildren = {};
        this.keyedRivals = {};
        this.activeExcess = null;
        this.activeSlot = null;
        if (parentRival.tagName === 'SLOT') { // Parent will "consume" a slot
            const slotName = parentRival.getAttribute('name') || '';
            this.activeSlot = this.slots[slotName] || null; // Mark active
            if (this.activeSlot) { // Children were specified for this slot!
                delete this.slots[slotName]; // (prevent "dupe slot" bug)
                this._setNextRival(null); // Move the cursor to the first elem
            }
        }
    }

    saveToStack() { // Creates an object copied with all cursor state
        this.instanceStack.push(Object.assign({ }, this)); // Copy to empty obj
    }

    loadFromStack() { // Remaining stack to "walk back" (non-recursive DFS)
        const stack = this.instanceStack;
        return stack.length > 0 && Object.assign(this, stack.pop());
    }

    loadFromExcessKeys() { // There were "excess", unmatched keyed elements
        if (!this.activeExcess) { // Convert needed appends and removes to array
            const child = Object.values(this.keyedChildren).map(v => [v, null]);
            const rival = Object.values(this.keyedRivals).map(v => [null, v]);
            this.activeExcess = rival.concat(child); // Do appends before remove
        }
        return this.activeExcess.length > 0; // Return true if there is any left
    }

    loadFromSlots() { // There are "excess" slots (copied, but deeply nested)
        const name = Object.keys(this.slots).pop(); // Get next ("pop" from obj)
        if (name === '' || name) { // Is name valid? (String of 0 or more)
            const sel = name ? `slot[name="${ name }"]` : 'slot:not([name])';
            const rivalSlot = this._rivalQuerySelector(sel);
            if (!rivalSlot) { // No slot (e.g., conditionally rendered, or typo)
                delete this.slots[name]; // (Ensure "consumed", if not init'ed)
                return this.loadFromSlots(); // If no elem, try popping again
            }
            this.initialize(this._querySelector(sel) || rivalSlot, rivalSlot);
            return true; // Indicate success: Child and rival slots are ready
        }
    }

    hasNext() {
        if (this.nextChild || this.nextRival || this.loadFromExcessKeys()) {
            return true; // Is pointing at another node, or has unmatched keys
        } else if (this.loadFromStack() || this.loadFromSlots()) { // Walk back
            return this.hasNext(); // Possibly loaded nodes nextChild, nextRival
        }
        return false; // Every load attempt is "false" (empty), end iteration
    }

    _setNextRival(rival) { // Traverse this.nextRival based on DOM or SLOT
        if (this.activeSlot !== null) { // Use activeSlot array for next instead
            if (this.activeSlot.length > 0) {
                this.nextRival = this.activeSlot.shift(); // Pop off next one
                this.nextRival._moduloIgnoreOnce = true; // Ensure no descend
            } else {
                this.nextRival = null;
            }
        } else {
            this.nextRival = rival ? rival.nextSibling : null; // Normal DOM traversal
        }
    }

    next() {
        let child = this.nextChild;
        let rival = this.nextRival;
        //if (!rival && !child && this.hasNext()) { return this.next(); } // TODO ALlow for calling w/o
hasNext
        if (!rival && this.activeExcess && this.activeExcess.length > 0) {
            return this.activeExcess.shift(); // Return the first pair
        }
        this.nextChild = child ? child.nextSibling : null;
        this._setNextRival(rival); // Traverse initially
        if (!this._disableKeys) { // Disable all "keyed element" logic
            let matchedRival = this.getMatchedNode(child, this.keyedChildren, this.keyedRivals);
            let matchedChild = this.getMatchedNode(rival, this.keyedRivals, this.keyedChildren);
            if (matchedRival === false) { // Child has a key, but does not match
                child = this.nextChild; // Simply ignore Child, and on to next
                this.nextChild = child ? child.nextSibling : null; // (1687)
            } else if (matchedChild === false) { // Rival has key, but no match
                rival = this.nextRival; // IGNORE rival - move on to
                this._setNextRival(rival); // (and setup next-next rival)
            }
            const keyWasFound = matchedRival !== null || matchedChild !== null;
            const matchFound = matchedChild !== child && keyWasFound;
            if (matchFound && matchedChild) { // Rival matches, but not child
                this.nextChild = child; // "Undo" this last "nextChild" (return)
                child = matchedChild; // Then substitute the matched instead
            }
            if (matchFound && matchedRival) {
                // Child matches, but not with rival. Swap in the correct one.
                this.modulo.assert(matchedRival !== rival, 'Dupe!');
                this.nextRival = rival; // "Undo" this last "nextRival"
                rival = matchedRival; // Then substitute the matched rival
            }
        }
        return [ child, rival ];
    }

    getMatchedNode(elem, keyedElems, keyedOthers) {
        const key = elem && elem.nodeType === 1 && elem.getAttribute('key');
        if (!key) {
            return null;
        } else if (key in keyedOthers) {
            const matched = keyedOthers[key];
            delete keyedOthers[key];
            return matched;
        } else if (key in keyedElems) {
            console.warn('MODULO: Duplicate key:', key);
        }
        keyedElems[key] = elem;
        return false;
    }
});

modulo.config.reconciler = {
    directiveShortcuts: [ [ /^@/, 'component.event' ],
                          [ /:$/, 'component.dataProp' ] ],
};
modulo.register('core', class Reconciler {
    constructor(modulo, def = {}) {
        this.modulo = modulo;
        this.directives = def.directives || {};
        this.directiveShortcuts = def.directiveShortcuts || [];
        if (this.directiveShortcuts.length === 0) { // TODO Remove this when tested
            this.directiveShortcuts = this.modulo.config.reconciler.directiveShortcuts;
        }
        this.patch = this.pushPatch;
        this.patches = [];
    }

    parseDirectives(rawName, directiveShortcuts) { //, foundDirectives) {
        if (/^[a-z0-9-]+$/i.test(rawName)) {
            return null; // if alphanumeric-only, stop right away
        }
        let name = rawName;
        for (const [ regexp, directive ] of directiveShortcuts) {
            if (rawName.match(regexp)) { // "Expand" shortcut into full version
                name = `[${directive}]` + name.replace(regexp, '');
            }
        }
        if (!name.startsWith('[')) {
            return null; // There are no directives, regular attribute, skip
        }
        // There are directives... time to resolve them
        const { cleanWord, stripWord } = this.modulo.registry.utils;
        const arr = [];
        const attrName = stripWord((name.match(/\][^\]]+$/) || [ '' ])[0]);
        for (const directiveName of name.split(']').map(cleanWord)) {
            // Skip the bare name itself, and filter for valid directives
            if (directiveName !== attrName) {// && directiveName in directives) {
                arr.push({ attrName, rawName, directiveName, name })
            }
        }
        return arr;
    }

    applyPatches(patches) {
        for (const patch of patches) { // Simply loop through given iterable
            this.applyPatch(patch[0], patch[1], patch[2], patch[3]);
        }
    }

    reconcileChildren(childParent, rivalParent, slots) {
        // Nonstandard nomenclature: "rival" is node we wish "child" to match
        const cursor = new this.modulo.registry.utils.DOMCursor(childParent, rivalParent, slots);
        while (cursor.hasNext()) {
            const [ child, rival ] = cursor.next();
            const needReplace = child && rival && (
                child.nodeType !== rival.nodeType ||
                child.nodeName !== rival.nodeName
            ); // Does this node to be swapped out? Swap if exist but mismatched

            if ((child && !rival) || needReplace) { // we have more rival, delete child
                this.patchAndDescendants(child, 'Unmount');
                this.patch(cursor.parentNode, 'removeChild', child);
            }

            if (needReplace) { // do swap with insertBefore
                this.patch(cursor.parentNode, 'insertBefore', rival, child.nextSibling);
                this.patchAndDescendants(rival, 'Mount');
            }

            if (!child && rival) { // we have less than rival, take rival
                // TODO: Possibly add directive resolution context to rival / child.originalChildren?
                this.patch(cursor.parentNode, 'appendChild', rival);
                this.patchAndDescendants(rival, 'Mount');
            }

            if (child && rival && !needReplace) {
                // Both exist and are of same type, let's reconcile nodes
                if (child.nodeType !== 1) { // text or comment node
                    if (child.nodeValue !== rival.nodeValue) { // update
                        this.patch(child, 'node-value', rival.nodeValue);
                    }
                } else if (!child.isEqualNode(rival)) { // sync if not equal
                    this.reconcileAttributes(child, rival);
                    if (rival.hasAttribute('modulo-ignore')) { // Don't descend
                        // console.log('Skipping ignored node');
                    } else if (child.isModulo) { // is a Modulo component
                        this.patch(child, 'rerender', rival);
                    } else { //} else if (!this.shouldNotDescend) {
                        cursor.saveToStack();
                        cursor.initialize(child, rival);
                    }
                }
            }
        }
    }

    pushPatch(node, method, arg, arg2 = null) {
        this.patches.push([ node, method, arg, arg2 ]);
    }
```

```
applyPatch(node, method, arg, arg2) { // take that, rule of 3!
    if (method === 'node-value') {
        node.nodeValue = arg;
    } else if (method === 'insertBefore') {
        node.insertBefore(arg, arg2); // Needs 2 arguments
    } else if (method.startsWith('directive-')) {
        method = method.substr(10);//'directive-'.length); // TODO: RM prefix (or generalizze)
        node[method].call(node, arg); // invoke directive method
    } else {
        node[method].call(node, arg); // invoke method
    }
}

patchDirectives(el, rawName, suffix, copyFromEl = null) {
    const foundDirectives = this.parseDirectives(rawName, this.directiveShortcuts);
    if (!foundDirectives || foundDirectives.length === 0) {
        return; // Fast route: "undefined" means no directives
    }
    const value = (copyFromEl || el).getAttribute(rawName); // Get value
    for (const directive of foundDirectives) {
        const dName = directive.directiveName; // e.g. "state.bind", "link"
        const fullName = dName + suffix; // e.g. "state.bindMount"
        const thisObj = this.directives[dName] || this.directives[fullName];
        if (thisObj) { // If a directive matches...
            const methodName = fullName.split('.')[1] || fullName;
            Object.assign(directive, { value, el });
            this.patch(thisObj, 'directive-' + methodName, directive);
        }
    }
}

reconcileAttributes(node, rival) {
    const myAttrs = new Set(node ? node.getAttributeNames() : []);
    const rivalAttributes = new Set(rival.getAttributeNames());

    // Check for new and changed attributes
    for (const rawName of rivalAttributes) {
        const attr = rival.getAttributeNode(rawName);
        if (myAttrs.has(rawName) && node.getAttribute(rawName) === attr.value) {
            continue; // Already matches, on to next
        }
        if (myAttrs.has(rawName)) { // If exists, trigger Unmount first
            this.patchDirectives(node, rawName, 'Unmount');
            }
            // Set attribute node, and then Mount based on rival value
            this.patch(node, 'setAttributeNode', attr.cloneNode(true));
            this.patchDirectives(node, rawName, 'Mount', rival);
        }

        // Check for old attributes that were removed (ignoring modulo- prefixed ones)
        for (const rawName of myAttrs) {
            if (!rivalAttributes.has(rawName) && !rawName.startsWith('modulo-')) {
                this.patchDirectives(node, rawName, 'Unmount');
                this.patch(node, 'removeAttribute', rawName);
            }
        }
    }

    patchAndDescendants(parentNode, actionSuffix) {
        if (parentNode.nodeType !== 1) {
            return; // Skip anything that isn't a regular HTML element
        }
        if (parentNode._moduloIgnoreOnce) {
            delete parentNode._moduloIgnoreOnce; // Ensure ignore is deleted
            return; // Skip ignored elements
        }
        const searchNodes = Array.from(parentNode.querySelectorAll('*'));
        for (const node of [ parentNode ].concat(searchNodes)) {
            for (const rawName of node.getAttributeNames()) { // Do patches
                this.patchDirectives(node, rawName, actionSuffix);
            }
        }
    }
});

modulo.register('util', function getAutoExportNames(contents) {
    // TODO: Change exports/Directives/etc to def processor to better expose
    const regexpG = /(function|class)\s+(\w+)/g;
    const regexp2 = /(function|class)\s+(\w+)/; // hack, refactor
    return (contents.match(regexpG) || []).map(s => s.match(regexp2)[2])
        .filter(s => s && !Modulo.INVALID_WORDS.has(s));
});

modulo.register('util', function showDevMenu() {
    const cmd = new URLSearchParams(window.location.search).get('mod-cmd');
    const rerun = `<h1><a href="?mod-cmd=${ cmd }">&#x27F3; ${ cmd }</a></h1>`;
    if (cmd) { // Command specified, skip dev menu, run, and replace HTML after
```

```
        const callback = () => { window.document.body.innerHTML = rerun; };
        const start = () => modulo.registry.commands[cmd](modulo, { callback });
        return window.setTimeout(start, modulo.config.commandDelay || 1000);
    } // Else: Display "COMMANDS:" menu in console
    const href = 'window.location.href += ';
    const font = 'font-size: 28px; padding: 0 8px 0 8px; border: 2px solid black;';
    const commandGetters = Object.keys(modulo.registry.commands).map(cmd =>
        ('get ' + cmd + ' () {' + href + '"?mod-cmd=' + cmd + '";}'));
    const clsCode = 'class COMMANDS {' + commandGetters.join('\n') + '}';
    new Function(`console.log('%c%', '${ font }', new (${ clsCode }))`)();
});

modulo.register('command', function build (modulo, opts = {}) {
    modulo.preprocessAndDefine(opts.callback, 'BuildCommand');
});

if (typeof window.moduloBuild === 'undefined') { // Not in a build, try loading
    if (typeof window.document !== 'undefined') { // Document exists
        modulo.loadString(modulo._DEVLIB_SOURCE, '_artifact'); // Load dev lib
        modulo.loadFromDOM(window.document.head, null, true); // Blocking load
        window.document.addEventListener('DOMContentLoaded', () => {
            modulo.loadFromDOM(window.document.body, null, true); // Async load
            modulo.preprocessAndDefine(modulo.registry.utils.showDevMenu);
        });
    } else if (typeof module !== 'undefined') { // Node.js
        module.exports = { modulo, window };
    }
}
```

ANSWERS

Category A: Data Reference Errors

1. **Uninitialized Variables:**
   Any variable used before being assigned a value would fall into this error. If the program contains references to variables without prior initialization, this would lead to runtime errors.
2. **Array Bound Violations:**
   Any array index that exceeds the defined array bounds would throw an error.

3. **Non-Integer Array Indices:**
   Ensure that any array index uses an integer value.
4. **Dangling Pointers:**
   If the program uses pointer arithmetic or references memory that has already been deallocated, it would cause a "dangling reference" error.
5. **Type Confusion in Aliased Memory Areas:**
   If the program uses type aliasing (like C's `union`) and refers to the same memory location as different types, ensure correct access to avoid data corruption.
6. **Data Type Mismatch:**
   The program might store data in one type (e.g., an integer) but expect another type (e.g., a string), which causes errors.

Errors Identified:

- Using variables before initialization.
- Index out of bounds errors in array references.
- Dangling pointers caused by improper memory management.

Category B: Data-Declaration Errors

1. **Missing Variable Declarations:**
   Variables that are referenced but not declared can cause runtime issues. Some languages require explicit declaration, and failure to do so leads to logical errors.
2. **Improper Default Attributes:**
   Implicitly initialized variables could be assigned the wrong types or default attributes, potentially causing unexpected behavior.
3. **Incorrect Variable Length/Type:**
   Variables might be declared with the wrong type (e.g., using `int` when `float` is required), leading to truncation or rounding errors.
4. **Confusing Variable Names:**
   Similar variable names (like `count` vs. `counts`) could lead to confusion, making the code harder to debug.

Errors Identified:

- Missing or incorrect variable declarations.
- Incorrect initialization types or lengths.

Category C: Computation Errors

1. **Mixed-Type Operations:**
   Combining types like integers and floating-point values can lead to data loss due to conversion.
2. **Overflow/Underflow Risks:**
   Operations resulting in values that exceed the variable's range (overflow) or fall below the range (underflow) could cause erratic behavior.
3. **Division by Zero:**
   If any division operation has the possibility of having zero as the divisor, it leads to runtime exceptions.
4. **Precision Loss in Floating-Point Operations:**
   Due to base-2 representation of floating-point numbers, operations might yield imprecise results.

Errors Identified:

- Division by zero.
- Overflow during calculations.
- Mixed-type arithmetic without proper casting.

Category D: Comparison Errors

1. **Incorrect Comparisons Between Data Types:**
   If the program compares variables of different types, like comparing a string to a number, errors could occur.
2. **Faulty Boolean Expressions:**
   Incorrect logical expressions, such as using `&` instead of `&&`, would result in the wrong evaluation of conditions.
3. **Precision Errors in Floating-Point Comparisons:**
   Floating-point numbers should be carefully compared due to potential precision issues.
4. **Operator Precedence Misunderstanding:**
   Assuming incorrect precedence (e.g., thinking `&&` takes precedence over `||` when it doesn't) leads to logical mistakes.

Errors Identified:

- Comparison of incompatible types (e.g., comparing a string and integer).
- Boolean expressions containing incorrect operators.

Category E: Control-Flow Errors

1. **Infinite Loops:**
   Loops without proper termination conditions can result in an infinite loop.
2. **Off-By-One Errors in Loops:**
   Improper handling of loop boundaries (e.g., starting at index `1` instead of `0` in a zero-indexed language) can cause extra or fewer iterations.
3. **Early Exit Conditions:**
   A loop might never run if its starting condition is false.

Errors Identified:

- Off-by-one errors in loops.
- Infinite loops.

Category F: Interface Errors

1. **Mismatch in Function Parameters:**
   The number, type, and order of arguments passed to functions must match their definitions.
2. **Global Variable Mismatch:**
   If the program uses global variables, ensure that they are declared consistently across different modules.

Errors Identified:

- Parameter mismatches between function calls and definitions.

Category G: Input/Output Errors

1. **File Handling Errors:**
   Files must be opened before use and closed after completion to avoid file descriptor leaks.
2. **Memory Allocation for Files:**
   Ensure that memory is allocated sufficiently to hold the contents of a file.

Errors Identified:

- Missing file close operations.

Category H: Other Checks

1. **Unused Variables:**
   Any variable that is declared but never used should be removed or utilized properly.
2. **Compilation Warnings:**
   Warnings generated during compilation should be investigated to prevent issues.

Errors Identified:

- Variables declared but not used.

Answers to Questions

1. **How many errors are there in the program? Mention the errors you have identified.**
   - There are multiple errors spread across all categories. Key errors include uninitialized variables, array index out of bounds, division by zero, parameter mismatches, infinite loops, and type mismatches.
2. **Which category of program inspection would you find more effective?**
   - **Category A: Data Reference Errors** is often the most effective, as uninitialized variables and memory issues can lead to critical program failures.
3. **Which type of error you are not able to identify using the program inspection?**
   - **Category H: Other Checks** like unused variables or certain performance-related issues may not be easy to identify with program inspection alone. These often require runtime checks or profiling.
4. **Is the program inspection technique worth applying?**
   - Yes, program inspection is highly valuable. It systematically reveals a wide range of potential bugs, from syntax errors to logical flaws, improving overall program quality and robustness.

Q2

1. Armstrong Number

A. Program Inspection

- Error: Incorrect computation of the remainder.
- Category: Computation Errors (Category C).
- Limitations: Doesn't identify debugging-related errors (e.g., breakpoints).

- Value: Effective for code structure and computation issues.

B. Debugging

- Error: Remainder computation issue.
- Fix: Set a breakpoint on the remainder computation line.

Corrected Code:
java

```java
public class Armstrong {
    public static void main(String args[]) {
        int num = Integer.parseInt(args[0]);
        int n = num;
        int check = 0, remainder;
        while (num > 0) {
            remainder = num % 10;
            check += (int) Math.pow(remainder, 3);
            num /= 10;
        }
        System.out.println(n + (check == n ? " is an Armstrong Number"
: " is not an Armstrong Number"));
    }
}
```

---

2. GCD and LCM

A. Program Inspection

- Errors:
  - GCD function: Incorrect loop condition.
  - LCM function: Logic error leading to an infinite loop.
- Category: Computation Errors (Category C).
- Limitations: Cannot identify runtime or logical errors.

B. Debugging

- Fix: Set breakpoints to verify execution.

Corrected Code:
java

```
import java.util.Scanner;

public class GCD_LCM {
    static int gcd(int x, int y) {
        int a = Math.max(x, y);
        int b = Math.min(x, y);
        while (b != 0) {
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }

    static int lcm(int x, int y) {
        return (x * y) / gcd(x, y);
    }

    public static void main(String args[]) {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter the two numbers: ");
        int x = input.nextInt();
        int y = input.nextInt();
        System.out.println("The GCD of two numbers is: " + gcd(x, y));
        System.out.println("The LCM of two numbers is: " + lcm(x, y));
        input.close();
    }
}
```

---

3. Knapsack

A. Program Inspection

- Error: Unintended increment of n in the option calculation.
- Category: Computation Errors (Category C).
- Limitations: Cannot identify runtime errors.

B. Debugging

- Fix: Set a breakpoint on the option calculation line.

Corrected Code:
java

```
public class Knapsack {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        int W = Integer.parseInt(args[1]);
        int[] profit = new int[N + 1];
        int[] weight = new int[N + 1];

        // Random item generation
        for (int n = 1; n <= N; n++) {
            profit[n] = (int) (Math.random() * 1000);
            weight[n] = (int) (Math.random() * W);
        }

        int[][] opt = new int[N + 1][W + 1];
        boolean[][] sol = new boolean[N + 1][W + 1];

        // Filling the DP table
        for (int n = 1; n <= N; n++) {
            for (int w = 1; w <= W; w++) {
                int option1 = opt[n - 1][w];
                int option2 = (weight[n] <= w) ? profit[n] + opt[n -
1][w - weight[n]] : Integer.MIN_VALUE;
                opt[n][w] = Math.max(option1, option2);
                sol[n][w] = (option2 > option1);
            }
        }

        // Print results
        System.out.println("Item\tProfit\tWeight\tTake");
        for (int n = 1; n <= N; n++) {
            System.out.println(n + "\t" + profit[n] + "\t" + weight[n]
+ "\t" + sol[n][W]);
        }
    }
}
```

---

4. Magic Number

A. Program Inspection

- Errors:
  - Inner loop condition should be `while (sum > 0)`.
  - Missing semicolons in calculations.
- Category: Computation Errors (Category C).

B. Debugging

- Fix: Set a breakpoint at the start of the inner loop.

Corrected Code:
java

```java
import java.util.*;

public class MagicNumberCheck {
    public static void main(String args[]) {
        Scanner ob = new Scanner(System.in);
        System.out.println("Enter the number to be checked:");
        int n = ob.nextInt();
        int sum = 0, num = n;

        while (num > 9) {
            sum = num;
            int s = 0;
            while (sum > 0) {
                s = s * (sum / 10);  // Corrected
                sum = sum % 10;      // Corrected
            }
            num = s;
        }

        System.out.println(n + (num == 1 ? " is a Magic Number." : "
is not a Magic Number."));
    }
}
```

---

5. Merge Sort

A. Program Inspection

- Errors:
  - Incorrect method calls in `mergeSort`.
  - Wrong implementation in `leftHalf` and `rightHalf` methods.
- Category: Computation Errors (Category C).

B. Debugging

- Fix: Set breakpoints to examine values during execution.

Corrected Code:
java

```
import java.util.*;

public class MergeSort {
    public static void main(String[] args) {
        int[] list = {14, 32, 67, 76, 23, 41, 58, 85};
        System.out.println("before: " + Arrays.toString(list));
        mergeSort(list);
        System.out.println("after: " + Arrays.toString(list));
    }

    public static void mergeSort(int[] array) {
        if (array.length > 1) {
            int[] left = leftHalf(array);
            int[] right = rightHalf(array);
            mergeSort(left);
            mergeSort(right);
            merge(array, left, right);
        }
    }

    public static int[] leftHalf(int[] array) {
        int size1 = array.length / 2;
        int[] left = Arrays.copyOfRange(array, 0, size1);
        return left;
    }

    public static int[] rightHalf(int[] array) {
        int size1 = array.length / 2;
        return Arrays.copyOfRange(array, size1, array.length);
    }

    public static void merge(int[] result, int[] left, int[] right) {
        int i1 = 0, i2 = 0;
        for (int i = 0; i < result.length; i++) {
            if (i2 >= right.length || (i1 < left.length && left[i1] <=
right[i2])) {
                result[i] = left[i1++];
            } else {
                result[i] = right[i2++];
            }
        }
    }
}
```

---

6. Multiply Matrices

A. Program Inspection

- Errors:
    - Loop indices should start from 0.
    - Incorrect error message.
- Category: Computation Errors (Category C).

B. Debugging

- Fix: Set breakpoints to inspect loop variable values.

Corrected Code:
java

B. Debugging

- Fix: Set breakpoints to inspect loop variable values.

Corrected Code:
java

```
import java.util.Scanner;

class MatrixMultiplication {
    public static void main(String args[]) {
        Scanner in = new Scanner(System.in);
        System.out.println("Enter rows and columns for the first
matrix:");
        int m = in.nextInt();
        int n = in.nextInt();
        int[][] first = new int[m][n];

        System.out.println("Enter elements of the first matrix:");
        for (int c = 0; c < m; c++)
            for (int d = 0; d < n; d++)
                first[c][d] = in.nextInt();

        System.out.println("Enter rows and columns for the second
matrix:");
        int p = in.nextInt();
        int q = in.nextInt();

        if (n != p) {
            System.out.println("Matrices with entered orders can't be
multiplied with each other.");
        } else {
            int[][] second = new int[p][q];
            int[][] multiply = new int[m][q];
            System.out.println("Enter elements of the second
matrix:");
            for (int c = 0; c < p; c++)
                for (int d = 0; d < q; d++)
                    second[c][d] = in.nextInt();

            for (int c = 0; c < m; c++) {
                for (int d = 0; d < q; d++) {
                    int sum = 0;
                    for (int k = 0; k < n; k++) {
                        sum += first[c][k] * second[k][d];
                    }
                    multiply[c][d] = sum;
                }
            }

            System.out.println("Product of entered matrices:");
            for (int c = 0; c < m; c++) {
                for (int d = 0; d < q; d++) {
                    System.out.print(multiply[c][d] + " ");
                }
                System.out.println();
            }
        }
    }

     }
```