```
enum LayoutType { defaultLayout, alternativeLayout }

class LayoutState {
  final Entity? selectedEntity;
  final LayoutType layoutType;

  LayoutState(
      {this.selectedEntity, this.layoutType = LayoutType.defaultLayout});

  LayoutState copyWith({Entity? selectedEntity, LayoutType? layoutType}) {
    return LayoutState(
      selectedEntity: selectedEntity ?? this.selectedEntity,
      layoutType: layoutType ?? this.layoutType,
    );
  }
}




class LayoutBloc extends Bloc<LayoutEvent, LayoutState> {
  LayoutBloc() : super(LayoutState()) {
    on<SelectEntityEvent>(_onSelectEntity);
    on<ToggleLayoutEvent>(_onToggleLayout);
  }

  void _onSelectEntity(SelectEntityEvent event, Emitter<LayoutState> emit) {
    emit(state.copyWith(selectedEntity: event.entity));
  }

  void _onToggleLayout(ToggleLayoutEvent event, Emitter<LayoutState> emit) {
    final newLayoutType = state.layoutType == LayoutType.defaultLayout
        ? LayoutType.alternativeLayout
        : LayoutType.defaultL

ayout;
    emit(state.copyWith(layoutType: newLayoutType));
  }
}



abstract class LayoutEvent {}

class SelectEntityEvent extends LayoutEvent {
  final Entity entity;

  SelectEntityEvent({required this.entity});
}

class ToggleLayoutEvent extends LayoutEvent {}



class LayoutInitial extends LayoutState {}



abstract class ThemeEvent {}

class ToggleThemeEvent extends ThemeEvent {}
```

```
class ChangeThemeEvent extends ThemeEvent {
  final ThemeData themeData;

  ChangeThemeEvent(this.themeData);
}




class ThemeState {
  final ThemeData themeData;
  final bool isDarkMode;

  ThemeState({required this.themeData, required this.isDarkMode});
}

class ThemeBloc extends Bloc<ThemeEvent, ThemeState> {
  ThemeBloc() : super(ThemeState(themeData: cliDarkTheme, isDarkMode: true)) {
    on<ToggleThemeEvent>((event, emit) {
      final isDarkMode = !state.isDarkMode;
      final currentThemeName = themes[isDarkMode ? 'light' : 'dark']!
          .keys
          .firstWhere(
              (name) =


>
                  themes[state.isDarkMode ? 'dark' : 'light']![name] ==
                  state.themeData,
              orElse: () => themes[isDarkMode ? 'dark' : 'light']!.keys.first);
      final newThemeData =
          themes[isDarkMode ? 'dark' : 'light']![currentThemeName]!;
      emit(ThemeState(themeData: newThemeData, isDarkMode: isDarkMode));
    });

    on<ChangeThemeEvent>((event, emit) {
      emit(
          ThemeState(themeData: event.themeData, isDarkMode: state.isDarkMode));
    });
  }
}



class EntriesSidebarWidget extends StatelessWidget {
  final Entities entries;

  EntriesSidebarWidget({
    required this.entries,
  });

  @override
  Widget build(BuildContext context) {
    return Container(
      width: 200,
      child: ListView.builder(
        itemCount: entries.length,
        itemBuilder: (context, index) {
          final entity = entries.elementAt(index);
          return ListTile(
            title: Text(entity.code),
            onTap: () {
```

```
      },
          );
        },
      ),
    );
  }
}



class ModelDetailScreen extends StatelessWidget {
  final Domain domain;
  final Model model;
  final List<String> path;
  final void Function(Entity entity) onEntitySelected;

  ModelDetailScreen({
    required this.domain,
    required this.model,
    required this.path,
    required this.onEntitySelected,
  });

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: HeaderWidget(
          path: path,
          onPathSegmentTapped: (index) {
            if (index == 0) {
              Navigator.popUntil(context, ModalRoute.withName('/'));
            } else if (index == 1) {
              Navigator.popUntil(context, ModalRoute.withName('/domain'));
            } else if (index == 2) {
              Navigator.pop(context);
            }
          },
          filters: [],
          onAddFilter: (FilterCriteria filter) {},
          onBookmark: () {},
        ),
      ),

    body: EntitiesWidget(
        entities: model.concepts,
        onEntitySelected: (entity) {
          onEntitySelected(entity);
          Navigator.push(
            context,
            MaterialPageRoute(
              builder: (context) => EntityDetailScreen(
                entity: entity,
              ),
            ),
          );
        },
        bookmarkManager: BookmarkManager(),
        onBookmarkCreated: (Bookmark bookmark) {},
      ),
    );
```

```
  }
}


class DomainsWidget extends StatelessWidget {
  final Domains domains;
  final void Function(Domain domain)? onDomainSelected;

  DomainsWidget({required this.domains, this.onDomainSelected});

  @override
  Widget build(BuildContext context) {
    return ListView.builder(
      itemCount: domains.length,
      itemBuilder: (context, index) {
        var domain = domains.elementAt(index);
        return ListTile(
          title: Text(domain.code),
          onTap: () {
            if (onDomainSelected != null) {

        onDomainSelected!(domain);
            }
          },
        );
      },
    );
  }
}



class DomainDetailScreen extends StatelessWidget {
  final Domain domain;
  final List<String> path;
  final void Function(Model model) onModelSelected;

  DomainDetailScreen({required this.domain, required this.onModelSelected})
      : path = ['Home', domain.code];

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: HeaderWidget(
          path: path,
          onPathSegmentTapped: (index) {
            if (index == 0) {
              Navigator.popUntil(context, ModalRoute.withName('/'));
            } else if (index == 1) {
              Navigator.pop(context);
            }
          },
          filters: [],
          onAddFilter: (FilterCriteria filter) {},
          onBookmark: () {},
        ),
      ),
      body: cms.ModelsWidget(
        models: domain.models,
```

```
        onModelSelected: (model) {
          onModelSelected(

model);
          Navigator.push(
            context,
            MaterialPageRoute(
              builder: (context) => ModelDetailScreen(
                domain: domain,
                model: model,
                path: path + [model.code],
                onEntitySelected: (entity) {
                },
              ),
            ),
          );
        },
      ),
    );
  }
}



class HomePage extends StatefulWidget {
  const HomePage({super.key, required this.title, required this.appLinks});

  final String title;
  final AppLinks appLinks;

  @override
  HomePageState createState() => HomePageState();
}

class HomePageState extends State<HomePage> {
  List<String> path = ['Home'];

  late IOneApplication app;
  Domain? selectedDomain;
  Model? selectedModel;
  Entities? selectedEntries;
  Entities? selectedEntities;
  Concept? selectedConcept;

  List<Bookmark> bookmarks = [];
  BookmarkManager bookmarkManager = BookmarkManager();

  bool showMetaCanvas = false;
  LayoutAlg

orithm _selectedAlgorithm = MasterDetailLayoutAlgorithm();
  Matrix4? _savedTransformation;

  @override
  void initState() {
    super.initState();
    initializeApp();
    widget.appLinks.uriLinkStream.listen(_handleBookmarkSelected);
  }

  void initializeApp() {
```

```
    app = OneApplication();

    if (app.groupedDomains.isNotEmpty) {
      selectedDomain = app.groupedDomains.first;

      if (selectedDomain!.models.isNotEmpty) {
        selectedModel = selectedDomain!.models.first;
        selectedEntries = selectedModel!.concepts;
      }
    }
  }

  void _handleDomainSelected(Domain domain) {
    setState(() {
      selectedDomain = domain;
      selectedModel = domain.models.isNotEmpty ? domain.models.first : null;
      selectedEntries = selectedModel?.concepts.isNotEmpty ?? false
          ? selectedModel!.concepts
          : null;
    });
  }

  void _handleModelSelected(Model model) {
    setState(() {
      selectedModel = model;
      selectedEntries =
          model.conce

pts.isNotEmpty ? model.getOrderedEntryConcepts() : null;
    });
  }

  void _handleBookmarkSelected(Uri? uri) {
    if (uri != null) {
    }
  }

  void _handleConceptSelected(Concept concept) {
    var domainModel = app.getDomainModels(selectedDomain!.codeFirstLetterLower,
        selectedModel!.codeFirstLetterLower);
    var modelEntries = domainModel.getModelEntries(concept.model.code);
    var entry = modelEntries?.getEntry(concept.code);
    setState(() {
      selectedConcept = concept;
      selectedEntities = entry;
    });
  }

  void _changeLayoutAlgorithm(LayoutAlgorithm algorithm) {
    setState(() {
      _savedTransformation ??= Matrix4.identity();
      _selectedAlgorithm = algorithm;
    });
  }

  void _saveTransformation(Matrix4 transformation) {
    setState(() {
      _savedTransformation = transformation;
    });
  }

  @override
```

```
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: buildAppBar(context),
      body: buildBody(context),
    );

 }

  AppBar buildAppBar(BuildContext context) {
    return AppBar(
      title: Row(
        children: [
          for (var domain in app.groupedDomains) buildDomainButton(domain),
          const Spacer(),
          ThemeDropdown(),
          buildIconButton(Icons.view_quilt, () {
            setState(() {
              showMetaCanvas = !showMetaCanvas;
            });
          }),
          buildIconButton(Icons.swap_horiz, () {
            context.read<LayoutBloc>().add(ToggleLayoutEvent());
          }),
          buildIconButton(Icons.brightness_6, () {
            context.read<ThemeBloc>().add(ToggleThemeEvent());
          }),
        ],
      ),
    );
  }

  Widget buildDomainButton(Domain domain) {
    return Padding(
      padding: const EdgeInsets.symmetric(horizontal: 8.0),
      child: GestureDetector(
        onTap: () => _handleDomainSelected(domain),
        child: Text(domain.code),
      ),
    );
  }

  IconButton buildIconButton(IconData icon, VoidCallback onPres

sed) {
    return IconButton(icon: Icon(icon), onPressed: onPressed);
  }

  Widget buildBody(BuildContext context) {
    return BlocProvider(
      create: (context) => LayoutBloc(),
      child: BlocBuilder<LayoutBloc, LayoutState>(
        builder: (context, state) {
          return showMetaCanvas
              ? buildMetaDomainCanvas()
              : buildLayoutTemplate();
        },
      ),
    );
  }

  MetaDomainCanvas buildMetaDomainCanvas() {
    final transitDomains = Domains();
```

```
    transitDomains.add(selectedDomain!);

    return MetaDomainCanvas(
      domains: transitDomains,
      initialTransformation: _savedTransformation,
      onTransformationChanged: _saveTransformation,
      onChangeLayoutAlgorithm: _changeLayoutAlgorithm,
      layoutAlgorithm: _selectedAlgorithm,
      decorators: [],
    );
  }

  Scaffold buildLayoutTemplate() {
    return Scaffold(
      appBar: AppBar(
        title: buildHeader(),
      ),
      body: Row(
        children: [
          bu

ildLeftSidebar(),
          buildMainContent(),
          buildRightSidebar(),
        ],
      ),
      bottomNavigationBar: const FooterWidget(),
    );
  }

  Widget buildLeftSidebar() {
    return Expanded(
      flex: 2,
      child: LeftSidebarWidget(
        entries: selectedEntries as Concepts,
        onConceptSelected: _handleConceptSelected,
      ),
    );
  }

  Widget buildMainContent() {
    return Expanded(
      flex: 8,
      child: selectedConcept != null
          ? MainContentWidget(
              entities: selectedEntities ?? Entities<Concept>(),
            )
          : const Text('No Concept selected'),
    );
  }

  Widget buildRightSidebar() {
    return selectedDomain != null
        ? Expanded(
            flex: 2,
            child: RightSidebarWidget(
              models: selectedDomain!.models,
              onModelSelected: _handleModelSelected,
            ),
          )
        : const Text('No Domain selected');
  }
```

```
  HeaderWidget buildHeader() {

   return HeaderWidget(
      filters: [],
      onAddFilter: (criteria) => print(criteria),
      onBookmark: () => print('onBookmark'),
      onPathSegmentTapped: print,
      path: path,
    );
  }
}

class ThemeDropdown extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final themeState = context.watch<ThemeBloc>().state;
    final brightness = themeState.isDarkMode ? 'dark' : 'light';
    final currentThemeName = themes[brightness]!.keys.firstWhere(
        (themeName) => themes[brightness]![themeName] == themeState.themeData,
        orElse: () => themes[brightness]!.keys.first);

    return DropdownButton<String>(
      value: currentThemeName,
      hint: Text('Select Theme'),
      items: themes[brightness]!.keys.map((String themeName) {
        return DropdownMenuItem<String>(
          value: themeName,
          child: Text(themeName),
        );
      }).toList(),
      onChanged: (themeName) {
        if (themeName != null) {
          final

themeData = themes[brightness]![themeName]!;
          context.read<ThemeBloc>().add(ChangeThemeEvent(themeData));
        }
      },
    );
  }
}




void main() {
}

class GraphApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Graph Visualization'),
        ),
        body: GraphWidget(),
      ),
    );
  }
}
```

```
class GraphWidget extends StatelessWidget {
  final Graph graph = Graph();

  GraphWidget() {
    OneApplication app = OneApplication();
    _buildGraph(app.groupedDomains);
  }

  void _buildGraph(Domains domains) {
    for (var domain in domains) {
      Node domainNode = Node.Id(domain.code);
      graph.addNode(domainNode);

      for (var model in domain.models) {
        Node modelNode = Node.Id(model.code);
        graph.addNode(modelNode);
        graph.addEdge(domainNode, modelNode);

        for (var concept in model.concepts) {
          Node c

onceptNode = Node.Id(concept.code);
          graph.addNode(conceptNode);
          graph.addEdge(modelNode, conceptNode);
        }
      }
    }
  }

  @override
  Widget build(BuildContext context) {
    final BuchheimWalkerConfiguration builder = BuchheimWalkerConfiguration()
      ..siblingSeparation = (100)
      ..levelSeparation = (150)
      ..subtreeSeparation = (150)
      ..orientation = BuchheimWalkerConfiguration.ORIENTATION_TOP_BOTTOM;

    return InteractiveViewer(
      constrained: false,
      boundaryMargin: EdgeInsets.all(100),
      minScale: 0.01,
      maxScale: 5.6,
      child: GraphView(
        graph: graph,
        algorithm: BuchheimWalkerAlgorithm(builder, TreeEdgeRenderer(builder)),
        builder: (Node node) {
          var nodeText = node.key!.value as String;
          return rectangleWidget(nodeText);
        },
      ),
    );
  }

  Widget rectangleWidget(String text) {
    return Container(
      padding: EdgeInsets.all(8),
      decoration: Box

Decoration(
        border: Border.all(color: Colors.black),
        borderRadius: BorderRadius.circular(4),
      ),
```

```
      child: Text(text),
    );
  }
}



class VisualizationGame extends FlameGame {
  @override
  Future<void> onLoad() async {
    super.onLoad();
    add(DomainComponent(label: 'Domain 1', position: Vector2(100, 100)));
    add(DomainComponent(label: 'Domain 2', position: Vector2(300, 100)));
    add(RelationComponent(start: Vector2(150, 150), end: Vector2(350, 150)));
  }
}

class DomainComponent extends PositionComponent {
  final String label;

  DomainComponent({required this.label, required Vector2 position}) {
    this.position = position;
    size = Vector2(200, 100);
  }

  @override
  void render(Canvas canvas) {
    super.render(canvas);
    final paint = Paint()..color = Colors.blue;
    canvas.drawRect(size.toRect(), paint);

    final textPainter = TextPainter(
      text: TextSpan(
        text: label,
        style: const TextStyle(color: Colors.white,

 fontSize: 14),
      ),
      textAlign: TextAlign.center,
      textDirection: TextDirection.ltr,
    );

    textPainter
      ..layout(maxWidth: size.x)
      ..paint(
        canvas,
        Offset((size.x - textPainter.width) / 2,
            (size.y - textPainter.height) / 2),
      );
  }
}

class RelationComponent extends Component {
  final Vector2 start;
  final Vector2 end;

  RelationComponent({required this.start, required this.end});

  @override
  void render(Canvas canvas) {
    final paint = Paint()
      ..color = Colors.black
```

```
      ..strokeWidth = 2.0;
    canvas.drawLine(start.toOffset(), end.toOffset(), paint);
  }

  @override
  void update(double dt) {}
}



class InitGame extends FlameGame {
  Entity entity;

  InitGame(this.entity);

  @override
  Future<void> onLoad() async {
    super.onLoad();

    String svgString = generateEntitySvg(entity);
    final svg = await Svg.loadFromString(svgString);

    final svgComponent = SvgComponent(
      svg: svg,

size: Vector2(400, 250),
      position: Vector2(0, 0),
    );

    add(svgComponent);
  }
}

void main() {
  Entity entity = Entity(
    name: "Sample Entity",
    code: "1234",
    attributes: {},
    parents: {},
    children: {},
  );

  runApp(GameWidget(game: InitGame(entity)));
}

String generateEntitySvg(Entity entity) {
  String svgTemplate =
      '''<svg width="400" height="250" xmlns="http://www.w3.org/2000/svg">
  <rect  x="10"  y="10"  width="380"  height="230"  fill="white"  stroke="black"
stroke-width="2" rx="10" ry="10"/>

  <!-- Image Placeholder -->
  <rect x="10" y="10" width="380" height="100" fill="#d6d6d6" rx="10" ry="10"/>
  <circle cx="50" cy="60" r="20" fill="rgba(0,0,0,0.1)"/>
  <polygon points="70,50 90,50 80,70" fill="rgba(0,0,0,0.1)"/>

  <!-- Title -->
  <text  x="20"  y="130"  font-family="Arial"  font-size="20"  fill="black">Name:
{{entityName}}</text>

  <!-- Code -->
  <text  x="20"  y="150"  font-family="Arial"  font-size="16"  fill="black">Code:
```

```
{{entityCode}

}</text>

  <!-- Attributes -->
  <text       x="20"       y="170"       font-family="Arial"       font-size="12"
fill="black">Attributes: {{attributes}}</text>

  <!-- Parents -->
  <text x="20" y="190" font-family="Arial" font-size="12" fill="black">Parents:
{{parents}}</text>

  <!-- Children -->
  <text x="20" y="210" font-family="Arial" font-size="12" fill="black">Children:
{{children}}</text>

  <!-- Action Button -->
  <rect x="320" y="210" width="70" height="20" fill="#6200ee" rx="5" ry="5"/>
  <text       x="335"       y="225"       font-family="Arial"       font-size="12"
fill="white">ACTION</text>
</svg>''';

  svgTemplate = svgTemplate.replaceFirst('{{entityName}}', entity.name);
  svgTemplate = svgTemplate.replaceFirst('{{entityCode}}', entity.code);
  svgTemplate =
      svgTemplate.replaceFirst('{{attributes}}', entity.attributes.toString());
  svgTemplate =
      svgTemplate.replaceFirst('{{parents}}', entity.parents.toString());
  svgTemplate =
      svgTemplate.replaceFirst('{{children}}', entity.children.toStr

ing());

  return svgTemplate;
}

class Entity {
  String name;
  String code;
  Map<String, dynamic> attributes;
  Map<String, dynamic> parents;
  Map<String, dynamic> children;

  Entity({
    required this.name,
    required this.code,
    required this.attributes,
    required this.parents,
    required this.children,
  });
}




class AlternativeLayout extends StatefulWidget {
  final Domains domains;
  final Entity? selectedEntity;
  final Function(Entity) onEntitySelected;

  const AlternativeLayout({
    Key? key,
```

```
    required this.domains,
    required this.selectedEntity,
    required this.onEntitySelected,
  }) : super(key: key);

  @override
  _AlternativeLayoutState createState() => _AlternativeLayoutState();
}

class _AlternativeLayoutState extends State<AlternativeLayout> {
  Offset _dragOffset = Offset.zero;

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onPanUpdate: (details) {
        setState(() {
          _dragOffset += detail

s.delta;
        });
      },
      child: Stack(
        children: [
          CustomPaint(
            size: Size.infinite,
            painter: RelationshipPainter(
                domains: widget.domains, offset: _dragOffset),
          ),
          Positioned(
            left: 16,
            top: 16,
            child: SingleChildScrollView(
              child: Container(
                width: 200, // Set a fixed width for the container
                child: Column(
                  crossAxisAlignment: CrossAxisAlignment.start,
                  children: [
                    ...widget.domains.toList().map((domain) {
                      return ListTile(
                        title: Text(domain.code),
                        onTap: () {
                        },
                      );
                    }).toList(),
                  ],
                ),
              ),
            ),
          ),
          if (widget.selectedEntity != null)
            Positioned(


              left: 200 + _dragOffset.dx,
              top: 200 + _dragOffset.dy,
              child: SizedBox(
                  width: 400,
                  height: 800,
                  child: EntityWidget(entity: widget.selectedEntity!)),
            ),
        ],
```

```
      ),
    );
  }
}

class RelationshipPainter extends CustomPainter {
  final Domains domains;
  final Offset offset;

  RelationshipPainter({required this.domains, required this.offset});

  @override
  void paint(Canvas canvas, Size size) {
    final paint = Paint()
      ..color = Colors.blue
      ..strokeWidth = 2;

    for (var domain in domains) {
      for (var model in domain.models) {
        for (var entity in model.concepts) {
          final entityPosition = Offset(
            100.0 * model.concepts.toList().indexOf(entity) + offset.dx,
            100.0 * domain.models.toList().indexOf(model) + offset.dy,
          );
          canvas.drawCircle(entityPosition, 5.0, paint);

          for (var relat

ion in entity.children) {
            final relatedEntityPosition = Offset(
              100.0 * entity.children.toList().indexOf(relation) + offset.dx,
              100.0 * domain.models.toList().indexOf(model) + offset.dy,
            );
            canvas.drawLine(entityPosition, relatedEntityPosition, paint);
          }
        }
      }
    }
  }

  @override
  bool shouldRepaint(covariant CustomPainter oldDelegate) {
    return true;
  }
}

class EntityCard extends StatelessWidget {
  final Entity entity;

  const EntityCard({Key? key, required this.entity}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Card(
      child: Padding(
        padding: const EdgeInsets.all(8.0),
        child: Column(
          mainAxisSize: MainAxisSize.min,
          children: [
            Text(entity.getStringFromAttribute('name') ?? 'Unnamed Entity'),
          ],
        ),
```

```
      ),
    );
  }
}



class MainContentWidget extends StatefulWidget {
  final E

ntities entities;

  MainContentWidget({super.key, required this.entities});

  @override
  State<MainContentWidget> createState() => _MainContentWidgetState();
}

class _MainContentWidgetState extends State<MainContentWidget> {
  Entity? selectedEntity;

  _handleEntitySelected(Entity entity) {
    setState(() {
      selectedEntity = entity;
    });
  }

  @override
  void initState() {
    super.initState();
    setState(() {
      selectedEntity = widget.entities.first as Entity;
    });
  }

  @override
  Widget build(BuildContext context) {
    return LayoutTemplate(
        header: Text('Filters of its children?'),
        leftSidebar: Column(
          children: widget.entities.toList().map((entity) {
            return ListTile(
              title: Text(getTitle(entity as Entity)),
              onTap: () => _handleEntitySelected(entity),
            );
          }).toList(),
        ),
        mainContent: Center(
          child: selectedEntity != null
              ? Entit

yWidget(entity: selectedEntity as Entity)
              : Text('Select an entity'),
        ),
        footer: Text('actions'),
        rightSidebar: Text('Entity navigation?'));
  }
}
```

```
class RightSidebarWidget extends StatelessWidget {
  final Models models;
  final void Function(Model model) onModelSelected;

  RightSidebarWidget({
    required this.models,
    required this.onModelSelected,
  });

  @override
  Widget build(BuildContext context) {
    return Container(
      width: 200,
      child: ListView.builder(
        itemCount: models.length,
        itemBuilder: (context, index) {
          final model = models.elementAt(index);
          return ListTile(
            title: Text(model.code),
            onTap: () => onModelSelected(model),
          );
        },
      ),
    );
  }
}



class ResponsiveLayout extends StatelessWidget {
  final Widget largeScreen;
  final Widget? mediumScreen;
  final Widget smallScreen;

  const ResponsiveLayout({
    Key? key,
    requir

ed this.largeScreen,
    this.mediumScreen,
    required this.smallScreen,
  }) : super(key: key);

  static int tabletBreakpoint = 768;
  static int desktopBreakpoint = 1440;

  static bool isSmallScreen(BuildContext context) {
    return MediaQuery.of(context).size.width < tabletBreakpoint;
  }

  static bool isMediumScreen(BuildContext context) {
    return MediaQuery.of(context).size.width >= tabletBreakpoint &&
        MediaQuery.of(context).size.width < desktopBreakpoint;
  }

  static bool isLargeScreen(BuildContext context) {
    return MediaQuery.of(context).size.width >= desktopBreakpoint;
  }

  @override
  Widget build(BuildContext context) {
    return LayoutBuilder(
```

```
      builder: (context, constraints) {
        if (constraints.maxWidth >= desktopBreakpoint) {
          return largeScreen;
        } else if (constraints.maxWidth >= tabletBreakpoint) {
          return mediumScreen ?? largeScreen;
        } else {
          return smallScreen;
        }
      },
    );

}
}



class FooterWidget extends StatelessWidget {
  const FooterWidget({super.key});

  @override
  Widget build(BuildContext context) {
    return Container(
      height: 60,
      child: Center(
        child: Text('Footer - ${DateTime.now().toString()}'),
      ),
    );
  }
}




class LayoutTemplate extends StatelessWidget {
  final Widget? header;
  final Widget? leftSidebar;
  final Widget? rightSidebar;
  final Widget mainContent;
  final Widget? footer;

  const LayoutTemplate({
    Key? key,
    this.header,
    this.leftSidebar,
    this.rightSidebar,
    required this.mainContent,
    this.footer,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return ResponsiveLayout(
      largeScreen: _buildLargeScreenLayout(),
      smallScreen: _buildSmallScreenLayout(),
    );
  }

  Widget _buildLargeScreenLayout() {
    return Column(
      children: [
        if (header != null) header!, // Remove _buildScrollableHeader
        Expanded(
```

```
 child: Row(
            children: [
              if (leftSidebar != null) Expanded(flex: 2, child: leftSidebar!),
              Expanded(flex: 6, child: mainContent),
              if (rightSidebar != null) Expanded(flex: 2, child: rightSidebar!),
            ],
          ),
        ),
        if (footer != null) footer!, // Remove _buildScrollableFooter
      ],
    );
  }

  Widget _buildSmallScreenLayout() {
    return Column(
      children: [
        if (header != null) header!,
        if (leftSidebar != null) leftSidebar!,
        Expanded(child: mainContent),
        if (rightSidebar != null) rightSidebar!,
        if (footer != null) footer!,
      ],
    );
  }

  Widget _buildScrollableHeader() {
    return ConstrainedBox(
      constraints: BoxConstraints(minWidth: double.infinity),
      child: header,
    );
  }

  Widget _buildScrollableFooter() {
    return ConstrainedBox(
      constraints: BoxConstraints(minWidth: double.infinity),
      child: footer,
    );
  }


 Widget _buildScrollableContent() {
    return ConstrainedBox(
      constraints: BoxConstraints(minHeight: double.infinity),
      child: mainContent,
    );
  }

  Widget _buildScrollableSidebar(Widget sidebar) {
    return ConstrainedBox(
      constraints: BoxConstraints(minHeight: double.infinity),
      child: sidebar,
    );
  }
}



class HeaderWidget extends StatelessWidget {
  final List<FilterCriteria> filters;
  final void Function(FilterCriteria filter) onAddFilter;
  final VoidCallback onBookmark;
```

```
  HeaderWidget({
    required this.filters,
    required this.onAddFilter,
    required this.onBookmark,
    required List<String> path,
    required void Function(dynamic index) onPathSegmentTapped,
  });

  @override
  Widget build(BuildContext context) {
    final theme = Theme.of(context);
    final colorScheme = theme.colorScheme;

    return Column(
      children: [
        Wrap(
          children: filters
              .map((filter) => Chip(
                    label:

Text(
                        '${filter.attribute}                  ${filter.operator}
${filter.value}'),
                    onDeleted: () {
                    },
                  ))
              .toList(),
        ),
      ],
    );
  }
}



class LeftSidebarWidget extends StatelessWidget {
  final Concepts entries;
  final void Function(Concept concept) onConceptSelected;

  LeftSidebarWidget({
    required this.entries,
    required this.onConceptSelected,
  });

  @override
  Widget build(BuildContext context) {
    return Container(
      width: 200,
      child: ListView.builder(
        itemCount: entries.length,
        itemBuilder: (context, index) {
          final concept = entries.elementAt(index);
          return ListTile(
            title: Text(concept.code),
            onTap: () => onConceptSelected(concept),
          );
        },
      ),
    );
  }
}
```

```
class HighlightDecorator implements UXDecorator {
  final Color color;
  final double thickness;

  HighlightDecorator({requ

ired this.color, this.thickness = 2.0});

  @override
  void apply(Canvas canvas, Offset position, double scale) {
    final paint = Paint()
      ..color = color
      ..style = PaintingStyle.stroke
      ..strokeWidth = thickness / scale;
    final rect = Rect.fromCenter(
        center: position, width: 110 / scale, height: 60 / scale);
    canvas.drawRect(rect, paint);
  }
}



abstract class UXDecorator {
  void apply(Canvas canvas, Offset position, double scale);
}




class TooltipDecorator implements UXDecorator {
  final String tooltip;
  final TextStyle textStyle;

  TooltipDecorator({required this.tooltip, required this.textStyle});

  @override
  void apply(Canvas canvas, Offset position, double scale) {
    final textSpan = TextSpan(text: tooltip, style: textStyle);
    final textPainter = TextPainter(
      text: textSpan,
      textDirection: TextDirection.ltr,
    );
    textPainter.layout();
    textPainter.paint(canvas, position + Offset(0, -50 / scale));
  }
}


enum

 LayoutAlgorithmType {
  forceDirected,
  grid,
  circular,
  masterDetail,
}



abstract class LayoutAlgorithm {
```

```
  Map<String, Offset> calculateLayout(Domains domains, Size size);
}



class GraphLayout {
  final Domains domains;
  final double defaultNodeWidth = 400;
  final double defaultNodeHeight = 600;

  GraphLayout({required this.domains});

  Graph buildGraph() {
    final graph = Graph();
    final domainNodes = <String, Node>{};

    for (var domain in domains) {
      final domainNode = Node.Id(domain.code);
      domainNodes[domain.code] = domainNode;
      graph.addNode(domainNode);

      for (var model in domain.models) {
        final modelNode = Node.Id(model.code);
        graph.addNode(modelNode);
        graph.addEdge(domainNode, modelNode);

        for (var entity in model.concepts) {
          final entityNode = Node.Id(entity.code);
          graph.addNode(entityNode);
          graph.addEdge(modelNode, entityNode);

          for (var attribute in entity.attr

ibutes) {
            final attributeNode = Node.Id(attribute.code);
            graph.addNode(attributeNode);
            graph.addEdge(entityNode, attributeNode);
          }

          for (var child in entity.children) {
            final childNode = Node.Id(child.code);
            graph.addNode(childNode);
            graph.addEdge(entityNode, childNode);
          }
        }
      }
    }
    return graph;
  }

  bool checkForCycles(Graph graph) {
    final visited = <Node>{};
    final stack = <Node>{};

    bool hasCycle(Node node) {
      if (stack.contains(node)) return true;
      if (visited.contains(node)) return false;
      visited.add(node);
      stack.add(node);

      for (final neighbor in graph.successorsOf(node)) {
```

```
        if (hasCycle(neighbor)) return true;
      }
      stack.remove(node);
      return false;
    }

    for (final node in graph.nodes) {
      if (hasCycle(node)) {
        return true;
      }
    }
    return false;
  }

  Map<String, Offset>

 calculateLayout(Size size) {
    final positions = <String, Offset>{};
    final random = Random();

    for (var domain in domains) {
      positions[domain.code] = Offset(
        random.nextDouble() * (size.width - defaultNodeWidth),
        random.nextDouble() * (size.height - defaultNodeHeight),
      );

      for (var model in domain.models) {
        positions[model.code] = Offset(
          random.nextDouble() * (size.width - defaultNodeWidth),
          random.nextDouble() * (size.height - defaultNodeHeight),
        );

        for (var entity in model.concepts) {
          positions[entity.code] = Offset(
            random.nextDouble() * (size.width - defaultNodeWidth),
            random.nextDouble() * (size.height - defaultNodeHeight),
          );

          for (var attribute in entity.attributes) {
            positions[attribute.code] = Offset(
              random.nextDouble() * (size.width - defaultNodeWidth),
              random.nextDouble() * (size.height - def

aultNodeHeight),
            );
          }

          for (var child in entity.children) {
            positions[child.code] = Offset(
              random.nextDouble() * (size.width - defaultNodeWidth),
              random.nextDouble() * (size.height - defaultNodeHeight),
            );
          }
        }
      }
    }

    return positions;
  }
}
```

```
class MetaDomainPainter extends CustomPainter {
  final Domains domains;
  final TransformationController transformationController;
  final LayoutAlgorithm layoutAlgorithm;
  final List<UXDecorator> decorators;
  final bool isDragging;
  final System system;
  final BuildContext context;
  final String? selectedNode;
  final Function(String) onNodeTap;

  MetaDomainPainter({
    required this.domains,
    required this.transformationController,
    required this.layoutAlgorithm,
    required this.decorators,
    required this.isDragging,
    required this.system,
    required this.context,
    required this.selectedNode,
    requir

ed this.onNodeTap,
  });

  Color _getColorForDomain(Domain domain, int level, double maxLevel) {
    double hue = (domain.hashCode % 360).toDouble();
    double saturation = 0.7;
    double brightness = (0.9 - (level / maxLevel) * 0.5).clamp(0.0, 1.0);
    return HSVColor.fromAHSV(1.0, hue, saturation, brightness).toColor();
  }

  @override
  void paint(Canvas canvas, Size size) {
    final positions = layoutAlgorithm.calculateLayout(domains, size);
    system.nodes.clear();

    double maxLevel = _calculateMaxLevel(domains);

    for (var domain in domains) {
      _createDomainNodes(domain, positions, 1, maxLevel);
    }

    system.render(canvas);

    system.renderText(canvas);
  }

  double _calculateMaxLevel(Domains domains) {
    double maxLevel = 1.0;
    for (var domain in domains) {
      for (var model in domain.models) {
        for (var concept in model.concepts) {
          maxLevel = max(maxLevel, _getConceptLevel(concept, 1));
        }
      }
    }
    return maxLev

el;
```

```
  }

  double _getConceptLevel(Concept concept, double currentLevel) {
    double maxLevel = currentLevel;
    for (var child in concept.children) {
      maxLevel = max(maxLevel, _getChildLevel(child, currentLevel + 1));
    }
    return maxLevel;
  }

  double _getChildLevel(Property child, double currentLevel) {
    double maxLevel = currentLevel;
    if (child is Child) {
      maxLevel = max(maxLevel,
          _getConceptLevel(child.destinationConcept, currentLevel + 1));
    }
    return maxLevel;
  }

  void _createDomainNodes(Domain domain, Map<String, Offset> positions,
      int level, double maxLevel) {
    final domainPosition = positions[domain.code];
    if (domainPosition == null) return;

    Color domainColor = _getColorForDomain(domain, level, maxLevel);
    Node domainNode = _createNode(domainPosition, domainColor, domain.code);
    system.addNode(domainNode);

    for (var model in domain.models) {
      final modelPosition = positions[model.code];
      if (mo

delPosition == null) continue;

      Color modelColor = _getColorForDomain(domain, level + 1, maxLevel);
      Node modelNode = _createNode(modelPosition, modelColor, model.code);
      system.addNode(modelNode);

      system.addNode(
          _createLineNode(domainPosition, modelPosition, 'has', 'belongs to'));

      for (var concept in model.concepts) {
        final conceptPosition = positions[concept.code];
        if (conceptPosition == null) continue;

        Color conceptColor = _getColorForDomain(domain, level + 2, maxLevel);
        Node conceptNode =
            _createNode(conceptPosition, conceptColor, concept.code);
        system.addNode(conceptNode);

        system.addNode(_createLineNode(
            modelPosition, conceptPosition, 'contains', 'is part of'));

        for (var child in concept.children) {
          final childPosition = positions[child.code];
          if (childPosition == null) continue;

          Color childColor = _getColorForDomain(domain, le

vel + 3, maxLevel);
          Node childNode = _createNode(childPosition, childColor, child.code);
          system.addNode(childNode);
```

```
          system.addNode(_createLineNode(conceptPosition, childPosition,
              child.code, (child as Neighbor).sourceConcept.code));
        }

        for (var parent in concept.parents) {
          final parentPosition = positions[parent.code];
          if (parentPosition != null) {
            system.addNode(_createLineNode(parentPosition, conceptPosition,
                parent.code, parent.sourceConcept.code));
          }
        }
      }
    }
  }

  Node _createLineNode(
      Offset start, Offset end, String fromToName, String toFromName) {
    Node node = Node();
    node.addComponent(LineComponent(
      start: start,
      end: end,
      fromToName: fromToName,
      toFromName: toFromName,
      fromTextStyle: Theme.of(context).textTheme.labelSmall!,
      toTextStyle: Theme.of(context).textTheme.labelSmall!,
    ));
    return

 node;
  }

  Node _createNode(Offset position, Color color, String label) {
    bool isSelected = label == selectedNode;
    Node node = Node();
    node.addComponent(RenderComponent(
      Paint()..color = color,
      Rect.fromCenter(center: position, width: 100, height: 50),
      glow: isSelected ? 10.0 : 0.0,
    ));
    node.addComponent(TextComponent(
      text: label,
      position: position,
      style:
          Theme.of(context).textTheme.labelLarge!.copyWith(color: Colors.white),
      backgroundColor: Colors.black.withOpacity(0.5),
    ));
    return node;
  }

  @override
  bool shouldRepaint(covariant CustomPainter oldDelegate) {
    return true;
  }
}




class MetaDomainCanvas extends StatefulWidget {
  final Domains domains;
  final LayoutAlgorithm layoutAlgorithm;
  final List<UXDecorator> decorators;
```

```
  final Matrix4? initialTransformation;
  final ValueChanged<Matrix4> onTransformationChanged;
  final ValueChanged<LayoutAlgorithm> onChangeLayoutAlgorithm;

  con

st MetaDomainCanvas({
    super.key,
    required this.domains,
    required this.layoutAlgorithm,
    required this.decorators,
    this.initialTransformation,
    required this.onTransformationChanged,
    required this.onChangeLayoutAlgorithm,
  });

  @override
  MetaDomainCanvasState createState() => MetaDomainCanvasState();
}

class MetaDomainCanvasState extends State<MetaDomainCanvas> {
  late TransformationController _transformationController;
  late LayoutAlgorithm _currentAlgorithm;
  bool _isDragging = false;
  late GameLoop _gameLoop;
  late System _system;
  late AnimationManager _animationManager;
  double _zoomLevel = 1.0;
  bool _isInitialLoad = true;
  String? _selectedNode;

  @override
  void initState() {
    super.initState();
    _transformationController = TransformationController();
    _currentAlgorithm = widget.layoutAlgorithm;
    _system = System();
    _animationManager = AnimationManager();
    _gameLoop = GameLoop(
      system: _system,
      animationMa

nager: _animationManager,
    );
    _gameLoop.start();

    WidgetsBinding.instance.addPostFrameCallback((_) {
      if (_isInitialLoad) {
        if (widget.initialTransformation != null) {
          _transformationController.value = widget.initialTransformation!;
          setState(() {
            _zoomLevel = _transformationController.value.getMaxScaleOnAxis();
          });
        } else {
          _centerAndZoom();
        }
        _isInitialLoad = false;
      }
    });

    _transformationController.addListener(() {
      widget.onTransformationChanged(_transformationController.value);
```

```
    });
  }

  void _onInteractionStart(ScaleStartDetails details) {
    setState(() {
      _isDragging = true;
    });
  }

  void _onInteractionEnd(ScaleEndDetails details) {
    setState(() {
      _isDragging = false;
    });
  }

  void _changeLayoutAlgorithm(LayoutAlgorithm algorithm) {
    setState(() {
      _currentAlgorithm = algorithm;
      widget.onChangeLayoutAlgorithm(algorith

m);
    });
  }

  void _zoom(double scaleFactor) {
    setState(() {
      _zoomLevel *= scaleFactor;
      _transformationController.value = Matrix4.identity()..scale(_zoomLevel);
    });
  }

  void _centerAndZoom() {
    final RenderBox renderBox = context.findRenderObject() as RenderBox;
    final Size canvasSize = renderBox.size;

    final layoutPositions =
        _currentAlgorithm.calculateLayout(widget.domains, canvasSize);
    final double minX = layoutPositions.values
        .map((offset) => offset.dx)
        .reduce((a, b) => a < b ? a : b);
    final double maxX = layoutPositions.values
        .map((offset) => offset.dx)
        .reduce((a, b) => a > b ? a : b);
    final double minY = layoutPositions.values
        .map((offset) => offset.dy)
        .reduce((a, b) => a < b ? a : b);
    final double maxY = layoutPositions.values
        .map((offset) => offset.dy)
        .reduce((a, b) => a > b ? a : b);

    final double graphWidth = maxX - minX;
    final double g

raphHeight = maxY - minY;

    final double scaleX =
        canvasSize.width / (graphWidth + 2 * 400); // Add some padding
    final double scaleY =
        canvasSize.height / (graphHeight + 2 * 400); // Add some padding
    final double scale = scaleX < scaleY ? scaleX : scaleY;

    final double offsetX =
```

```
        (canvasSize.width - graphWidth * scale) / 2 - minX * scale;
    final double offsetY =
        (canvasSize.height - graphHeight * scale) / 2 - minY * scale;

    _transformationController.value = Matrix4.identity()
      ..translate(offsetX, offsetY)
      ..scale(scale);

    setState(() {
      _zoomLevel = scale;
    });
  }

  void _onNodeTap(String nodeCode) {
    setState(() {
      _selectedNode = nodeCode;
    });
  }

  void _handleTap(TapUpDetails details) {
    final RenderBox renderBox = context.findRenderObject() as RenderBox;
    final tapPosition = _transformationController
        .toScene(renderBox.globalToLocal(details.globalPosition));

    final layoutP

ositions =
        _currentAlgorithm.calculateLayout(widget.domains, renderBox.size);

    const double margin = 10.0; // Adjust the margin size as needed

    for (var entry in layoutPositions.entries) {
      final nodeRect = Rect.fromCenter(
          center: entry.value,
          width: 100 + margin * 2,
          height: 50 + margin * 2);
      if (nodeRect.contains(tapPosition)) {
        _onNodeTap(entry.key);
        break;
      }
    }
  }

  @override
  Widget build(BuildContext context) {
    return Stack(
      children: [
        Column(
          children: [
            Row(
              mainAxisAlignment: MainAxisAlignment.start,
              children: [
                LayoutAlgorithmIcon(
                  icon: Icons.auto_fix_high,
                  name: 'Force Directed',
                  onTap: () =>
                      _changeLayoutAlgorithm(ForceDirectedLayoutAlgorithm()),
                  isActive: _currentAlgorithm is ForceDirectedLayoutAlgorithm,

          ),
                LayoutAlgorithmIcon(
                  icon: Icons.grid_on,
```

```
                  name: 'Grid',
                  onTap: () => _changeLayoutAlgorithm(GridLayoutAlgorithm()),
                  isActive: _currentAlgorithm is GridLayoutAlgorithm,
                ),
                LayoutAlgorithmIcon(
                  icon: Icons.circle,
                  name: 'Circular',
                  onTap: () =>
                      _changeLayoutAlgorithm(CircularLayoutAlgorithm()),
                  isActive: _currentAlgorithm is CircularLayoutAlgorithm,
                ),
                LayoutAlgorithmIcon(
                  icon: Icons.format_indent_increase,
                  name: 'Master Detail',
                  onTap: () =>
                      _changeLayoutAlgorithm(MasterDetailLayoutAlgorithm()),
                  isActive: _currentAlgorithm is MasterDetailLayoutAlgorithm,
                ),
                LayoutAlgorithmIcon(
                  icon: Icons.

account_tree,
                  name: 'Ranked Tree',
                  onTap: () =>
                      _changeLayoutAlgorithm(RankedEmbeddingLayoutAlgorithm()),
                  isActive: _currentAlgorithm is RankedEmbeddingLayoutAlgorithm,
                ),
              ],
            ),
            Expanded(
              child: GestureDetector(
                onScaleStart: _onInteractionStart,
                onScaleEnd: _onInteractionEnd,
                onTapUp: _handleTap,
                child: InteractiveViewer(
                  transformationController: _transformationController,
                  onInteractionUpdate: (details) {
                    setState(() {
                      _transformationController.value =
                          _transformationController.value
                            ..translate(details.focalPointDelta.dx,
                                details.focalPointDelta.dy)
                            ..scale(details.scale);

  });
                  },
                  minScale: 0.1,
                  maxScale: 5.0,
                  child: CustomPaint(
                    size: Size.infinite,
                    painter: MetaDomainPainter(
                      domains: widget.domains,
                      transformationController: _transformationController,
                      layoutAlgorithm: _currentAlgorithm,
                      decorators: [],
                      isDragging: _isDragging,
                      system: _system,
                      context: context,
                      selectedNode: _selectedNode,
                      onNodeTap: _onNodeTap,
                    ),
                  ),
```

```
                ),
              ),
            ),
          ],
        ),
        Positioned(
          bottom: 16.0,
          right: 16.0,
          child: Row(
            children: [
              FloatingActionButton(
                onPressed: () => _zoom(1.1),
                background

Color: Colors.transparent,
                elevation: 0,
                child: const Icon(Icons.add, color: Colors.white),
              ),
              const SizedBox(width: 16.0),
              FloatingActionButton(
                onPressed: () => _zoom(0.9),
                backgroundColor: Colors.transparent,
                elevation: 0,
                child: const Icon(Icons.remove, color: Colors.white),
              ),
              const SizedBox(width: 16.0),
              FloatingActionButton(
                onPressed: _centerAndZoom,
                backgroundColor: Colors.transparent,
                elevation: 0,
                child:
                    const Icon(Icons.center_focus_strong, color: Colors.white),
              ),
              const SizedBox(width: 16.0),
              Container(
                padding: const EdgeInsets.all(8.0),
                decoration: BoxDecoration(
                  color: Colors.black54,
                  borderRadius: Bord

erRadius.circular(4.0),
                ),
                child: Text(
                  'Zoom: ${(_zoomLevel * 100).toInt()}%',
                  style: const TextStyle(color: Colors.white),
                ),
              ),
            ],
          ),
        ),
      ],
    );
  }

  @override
  void dispose() {
    _gameLoop.stop();
    super.dispose();
  }
}
```

```
class RankedEmbeddingLayoutAlgorithm extends LayoutAlgorithm {
  final double nodeWidth = 100.0;
  final double nodeHeight = 50.0;
  final double verticalGap = 80.0;
  final double horizontalGap = 30.0;

  @override
  Map<String, Offset> calculateLayout(Domains domains, Size size) {
    final positions = <String, Offset>{};
    final domainSizes = <String, Size>{};

    for (var domain in domains) {
      final domainSize = _calculateDomainSize(domain);
      domainSizes[domain.code] = domainSize;
    }

    double currentX = 0.0;
    for (var domain in domains) {
      final domainSize = domainSizes[domain.code]!;

   final rootX = currentX + domainSize.width / 2;
      final rootY = domainSize.height / 2 + verticalGap;
      final root = TreeNode(domain.code, Offset(rootX, rootY));
      positions[domain.code] = root.position;
      _calculateModelPositions(root, domain.models, currentX,
          currentX + domainSize.width, positions);
      currentX += domainSize.width + horizontalGap;
    }

    return positions;
  }

  Size _calculateDomainSize(Domain domain) {
    double maxWidth = 0.0;
    double totalHeight = 0.0;

    for (var model in domain.models) {
      final modelSize = _calculateModelSize(model);
      maxWidth = max(maxWidth, modelSize.width);
      totalHeight += modelSize.height + verticalGap;
    }

    return Size(maxWidth, totalHeight);
  }

  Size _calculateModelSize(Model model) {
    double maxWidth = 0.0;
    double totalHeight = 0.0;

    for (var concept in model.concepts) {
      final conceptSize = _calculateConceptSize(concept);
      maxWidth = max(maxWidth, conce

ptSize.width);
      totalHeight += conceptSize.height + verticalGap;
    }

    return Size(maxWidth, totalHeight);
  }
```

```
  Size _calculateConceptSize(Concept concept) {
    double maxWidth = 0.0;
    double totalHeight = 0.0;

    for (var child in concept.children) {
      final childSize =
          _calculateConceptSize((child as Child).destinationConcept);
      maxWidth = max(maxWidth, childSize.width);
      totalHeight += childSize.height + verticalGap;
    }

    for (var attribute in concept.attributes) {
      maxWidth = max(maxWidth, nodeWidth);
      totalHeight += nodeHeight + verticalGap;
    }

    return Size(maxWidth, totalHeight);
  }

  void _calculateModelPositions(TreeNode parent, Models models, double xMin,
      double xMax, Map<String, Offset> positions) {
    if (models.isEmpty) return;

    final y = parent.position.dy + verticalGap;
    final width = (xMax - xMin) / max(1, models.length);

    for (var i = 0; i < models.length; i++) {
      final model = mod

els.at(i);
      final x = xMin + i * width + width / 2;
      final childNode = TreeNode(model.code, Offset(x, y));
      parent.children.add(childNode);
      positions[childNode.key] = childNode.position;
      _calculateConceptPositions(childNode, model.concepts, xMin + i * width,
          xMin + (i + 1) * width, positions);
    }
  }

  void _calculateConceptPositions(TreeNode parent, Concepts concepts,
      double xMin, double xMax, Map<String, Offset> positions) {
    if (concepts.isEmpty) return;

    final y = parent.position.dy + verticalGap;
    final width = (xMax - xMin) / max(1, concepts.length);

    for (var i = 0; i < concepts.length; i++) {
      final concept = concepts.at(i);
      final x = xMin + i * width + width / 2;
      final childNode = TreeNode(concept.code, Offset(x, y));
      parent.children.add(childNode);
      positions[childNode.key] = childNode.position;
      _calculateConceptChildrenPositions(childNode, concept.children,
          xMin + i * wid

th, xMin + (i + 1) * width, positions);
      _calculateAttributePositions(childNode, concept.attributes,
          xMin + i * width, xMin + (i + 1) * width, positions);
    }
  }

  void _calculateConceptChildrenPositions(TreeNode parent, Children children,
```

```
      double xMin, double xMax, Map<String, Offset> positions) {
    if (children.isEmpty) return;

    final y = parent.position.dy + verticalGap;
    final width = (xMax - xMin) / max(1, children.length);

    for (var i = 0; i < children.length; i++) {
      final child = children.at(i);
      final x = xMin + i * width + width / 2;
      final childNode = TreeNode(child.code, Offset(x, y));
      parent.children.add(childNode);
      positions[childNode.key] = childNode.position;

      if (child is Child && child.navigate) {
        _calculateConceptChildrenPositions(
            childNode,
            child.destinationConcept.children,
            xMin + i * width,
            xMin + (i + 1) * width,
            positions);

     }
    }
  }

  void _calculateAttributePositions(TreeNode parent, Attributes attributes,
      double xMin, double xMax, Map<String, Offset> positions) {
    if (attributes.isEmpty) return;

    final y = parent.position.dy + verticalGap;
    final width = (xMax - xMin) / max(1, attributes.length);

    for (var i = 0; i < attributes.length; i++) {
      final attribute = attributes.at(i);
      final x = xMin + i * width + width / 2;
      final childNode = TreeNode(attribute.code, Offset(x, y));
      parent.children.add(childNode);
      positions[childNode.key] = childNode.position;
    }
  }
}


class Graph {
  Map<String, List<String>> adjacencyList = {};

  void addEdge(String u, String v) {
    adjacencyList.putIfAbsent(u, () => []).add(v);
    adjacencyList.putIfAbsent(v, () => []).add(u);
  }

  List<String>? getNeighbors(String u) {
    return adjacencyList[u];
  }
}




class GraphTraversal {
  final Graph graph;
```

```
  GraphTraversal(this.graph);

  List<String> bfs(Strin

g start) {
    List<String> visited = [];
    Queue<String> queue = Queue();
    queue.add(start);

    while (queue.isNotEmpty) {
      String node = queue.removeFirst();
      if (!visited.contains(node)) {
        visited.add(node);
        graph.getNeighbors(node)?.forEach((neighbor) {
          if (!visited.contains(neighbor)) {
            queue.add(neighbor);
          }
        });
      }
    }
    return visited;
  }

  List<String> dfs(String start) {
    List<String> visited = [];
    _dfsHelper(start, visited);
    return visited;
  }

  void _dfsHelper(String node, List<String> visited) {
    if (visited.contains(node)) return;
    visited.add(node);
    graph.getNeighbors(node)?.forEach((neighbor) {
      if (!visited.contains(neighbor)) {
        _dfsHelper(neighbor, visited);
      }
    });
  }
}




class CircularLayoutAlgorithm extends LayoutAlgorithm {
  final double nodeWidth;
  final double nodeHeight;
  final double levelDistanceIncrement;

  CircularLayoutAlgo

rithm({
    this.nodeWidth = 600.0,
    this.nodeHeight = 300.0,
    this.levelDistanceIncrement = 200.0,
  });

  @override
  Map<String, Offset> calculateLayout(Domains domains, Size size) {
    final positions = <String, Offset>{};
    final center = Offset(size.width / 2, size.height / 2);
```

```
    if (domains.isEmpty) {
      return positions;
    }

    final domain = domains.first; // Only consider the first domain

    final requiredSpace = _calculateDomainSpace(domain);

    _positionRoot(domain, center, positions, requiredSpace);

    return positions;
  }

  double _calculateDomainSpace(Domain domain) {
    double maxSpace = nodeWidth;
    for (var model in domain.models) {
      maxSpace = max(maxSpace, _calculateModelSpace(model));
    }
    return maxSpace;
  }

  double _calculateModelSpace(Model model) {
    double maxSpace = nodeWidth;
    final entryConcepts =
        model.concepts.where((concept) => concept.entry).toList();
    for (var concept in entryConcepts) {

maxSpace = max(maxSpace, _calculateConceptSpace(concept));
    }
    return maxSpace;
  }

  double _calculateConceptSpace(Concept concept) {
    double maxSpace = nodeWidth;
    for (var child in concept.children.whereType<Child>()) {
      maxSpace =
          max(maxSpace, _calculateConceptSpace(child.destinationConcept));
    }
    return maxSpace;
  }

  void _positionRoot(Domain domain, Offset center,
      Map<String, Offset> positions, double requiredSpace) {
    final rootRadius = requiredSpace / (0.5 * pi);

    positions[domain.code] = center;

    _positionModels(domain, center, positions, rootRadius, 1, 0, 4 * pi);
  }

  void _positionModels(
      Domain domain,
      Offset parentPosition,
      Map<String, Offset> positions,
      double levelDistance,
      int level,
      double startAngle,
      double angleRange) {
    final models = domain.models.toList();
    if (models.isEmpty) return;

    double currentAngle = startAngle;
```

```
    for (var model in models) {

  final angleStep = angleRange / models.length;
      final modelPosition = parentPosition +
          Offset(levelDistance * cos(currentAngle + angleStep / 2),
              levelDistance * sin(currentAngle + angleStep / 2));
      positions[model.code] = modelPosition;

      _positionConcepts(
          model,
          modelPosition,
          positions,
          levelDistance + levelDistanceIncrement,
          level + 1,
          currentAngle,
          angleStep);

      currentAngle += angleStep;
    }
  }

  void _positionConcepts(
      Model model,
      Offset parentPosition,
      Map<String, Offset> positions,
      double levelDistance,
      int level,
      double startAngle,
      double angleRange) {
    final concepts = model.concepts.where((concept) => concept.entry).toList();
    if (concepts.isEmpty) return;

    double currentAngle = startAngle;
    for (var concept in concepts) {
      final angleStep = angleRange / concepts.length;
      final conceptPositio

n = parentPosition +
          Offset(levelDistance * cos(currentAngle + angleStep / 2),
              levelDistance * sin(currentAngle + angleStep / 2));
      positions[concept.code] = conceptPosition;

      _positionConceptChildren(
          concept,
          conceptPosition,
          positions,
          levelDistance + levelDistanceIncrement,
          level + 1,
          currentAngle,
          angleStep);

      currentAngle += angleStep;
    }
  }

  void _positionConceptChildren(
      Concept concept,
      Offset parentPosition,
      Map<String, Offset> positions,
      double levelDistance,
      int level,
      double startAngle,
```

```
      double angleRange) {
    final childNodes = concept.children.whereType<Child>().toList();
    if (childNodes.isNotEmpty) {
      _positionChildNodes(childNodes, parentPosition, positions, levelDistance,
          level, startAngle, angleRange);
    }

    for (var attribute in concept.attributes.whereType<Attribute>()) {
      _positi

onAttribute(attribute, parentPosition, positions);
    }
  }

  void _positionChildNodes(
      List<Child> children,
      Offset parentPosition,
      Map<String, Offset> positions,
      double levelDistance,
      int level,
      double startAngle,
      double angleRange) {
    double currentAngle = startAngle;
    for (var child in children) {
      final angleStep = angleRange / children.length;
      final childPosition = parentPosition +
          Offset(levelDistance * cos(currentAngle + angleStep / 2),
              levelDistance * sin(currentAngle + angleStep / 2));
      positions[child.destinationConcept.code] = childPosition;

      currentAngle += angleStep;
    }
  }

  void _positionAttribute(Attribute attribute, Offset parentPosition,
      Map<String, Offset> positions) {
    final attributePosition = parentPosition +
        Offset(-nodeWidth, 0); // Example positioning for attributes
    positions[attribute.code] = attributePosition;
  }
}




class RadialTreeLa

youtAlgorithm extends LayoutAlgorithm {
  final double nodeWidth;
  final double nodeHeight;
  final double levelGap;

  RadialTreeLayoutAlgorithm({
    this.nodeWidth = 100.0,
    this.nodeHeight = 50.0,
    this.levelGap = 50.0,
  });

  @override
  Map<String, Offset> calculateLayout(Domains domains, Size size) {
    final positions = <String, Offset>{};
```

```
    double centerX = size.width / 2;
    double centerY = size.height / 2;
    double angleStep = 2 * pi / domains.length;
    double currentAngle = 0.0;

    for (var domain in domains) {
      _calculatePositionsForDomain(domain.code, domain, centerX, centerY, 0,
          currentAngle, angleStep, positions);
      currentAngle += angleStep;
    }

    return positions;
  }

  void _calculatePositionsForDomain(
      String nodeId,
      Domain domain,
      double centerX,
      double centerY,
      int level,
      double angle,
      double angleStep,
      Map<String, Offset> positions) {
    double radius = level * levelGap

;
    double x = centerX + radius * cos(angle);
    double y = centerY + radius * sin(angle);

    positions[nodeId] = Offset(x, y);

    double childAngleStep = angleStep / max(domain.models.length, 1);
    double currentChildAngle =
        angle - (childAngleStep * (domain.models.length - 1)) / 2;

    for (var model in domain.models) {
      _calculatePositionsForModel(model.code, model, x, y, level + 1,
          currentChildAngle, childAngleStep, positions);
      currentChildAngle += childAngleStep;
    }
  }

  void _calculatePositionsForModel(
      String modelId,
      Model model,
      double centerX,
      double centerY,
      int level,
      double angle,
      double angleStep,
      Map<String, Offset> positions) {
    double radius = level * levelGap;
    double x = centerX + radius * cos(angle);
    double y = centerY + radius * sin(angle);

    positions[modelId] = Offset(x, y);

    double childAngleStep = angleStep / max(model.concepts.length, 1);
    double cur

rentChildAngle =
        angle - (childAngleStep * (model.concepts.length - 1)) / 2;
```

```
  for (var entity in model.concepts) {
    _calculatePositionsForEntity(entity.code, model, entity, x, y, level + 1,
        currentChildAngle, childAngleStep, positions);
    currentChildAngle += childAngleStep;
  }
}

void _calculatePositionsForEntity(
    String entityId,
    Model model,
    Entity entity,
    double centerX,
    double centerY,
    int level,
    double angle,
    double angleStep,
    Map<String, Offset> positions) {
  double radius = level * levelGap;
  double x = centerX + radius * cos(angle);
  double y = centerY + radius * sin(angle);

  positions[entityId] = Offset(x, y);

  final safeEntity = Concept.safeGetConcept(model, entity);

  double childAngleStep =
      angleStep / max(safeEntity.concept.children.length, 1);
  double currentChildAngle =
      angle - (childAngleStep * (safeEntity.concept.children.length - 1)) / 2;



  for (var child in safeEntity.concept.children) {
    _calculatePositionsForConceptChild(child.code, child as Child, x, y,
        level + 1, currentChildAngle, childAngleStep, positions);
    currentChildAngle += childAngleStep;
  }
}

void _calculatePositionsForConceptChild(
    String childId,
    Child child,
    double centerX,
    double centerY,
    int level,
    double angle,
    double angleStep,
    Map<String, Offset> positions) {
  double radius = level * levelGap;
  double x = centerX + radius * cos(angle);
  double y = centerY + radius * sin(angle);

  positions[childId] = Offset(x, y);

  double childAngleStep =
      angleStep / max(child.destinationConcept.children.length, 1);
  double currentChildAngle = angle -
      (childAngleStep * (child.destinationConcept.children.length - 1)) / 2;

  for (var grandChild in child.destinationConcept.children) {
    _calculatePositionsForConceptChild(grandChild.code, grandChild
```

```
 as Child,
          x, y, level + 1, currentChildAngle, childAngleStep, positions);
      currentChildAngle += childAngleStep;
    }
  }
}



class NetworkFlowLayoutAlgorithm extends LayoutAlgorithm {
  @override
  Map<String, Offset> calculateLayout(Domains domains, Size size) {
    final positions = <String, Offset>{};
    final graph = <String, Map<String, double>>{};

    for (var domain in domains) {
      positions[domain.code] = Offset(size.width / 2, size.height / 2);
      graph[domain.code] = {};

      for (var model in domain.models) {
        final modelPosition = Offset(size.width * 0.25, size.height * 0.25);
        positions[model.code] = modelPosition;
        graph[domain.code]![model.code] =
            _distance(positions[domain.code]!, modelPosition);

        for (var entity in model.concepts) {
          final entityPosition = Offset(size.width * 0.75, size.height * 0.75);
          positions[entity.code] = entityPosition;
          graph[model.code]![entity.co

de] =
              _distance(modelPosition, entityPosition);

          for (var child in entity.children) {
            final childPosition = Offset(size.width * 0.5, size.height * 0.5);
            positions[child.code] = childPosition;
            graph[entity.code]![child.code] =
                _distance(entityPosition, childPosition);
          }
        }
      }
    }

    final maxFlow = _edmondsKarp(graph, domains.first.code, domains.last.code);

    return positions;
  }

  double _distance(Offset a, Offset b) {
    return (a - b).distance;
  }

  double _edmondsKarp(
      Map<String, Map<String, double>> graph, String source, String sink) {
    final residualGraph = <String, Map<String, double>>{};
    for (var u in graph.keys) {
      residualGraph[u] = {};
      for (var v in graph[u]!.keys) {
        residualGraph[u]![v] = graph[u]![v]!;
      }
    }
```

```
    final parent = <String?, String?>{};
    double maxFlow = 0;

    bool bfs(String source, String sink) {
      fin

al visited = <String>{};
      final queue = Queue<String>();
      queue.add(source);
      visited.add(source);
      parent[source] = null;

      while (queue.isNotEmpty) {
        final u = queue.removeFirst();
        for (var v in residualGraph[u]!.keys) {
          if (!visited.contains(v) && residualGraph[u]![v]! > 0) {
            queue.add(v);
            visited.add(v);
            parent[v] = u;
            if (v == sink) return true;
          }
        }
      }
      return false;
    }

    while (bfs(source, sink)) {
      double pathFlow = double.infinity;
      for (var v = sink; v != source; v = parent[v]!) {
        final u = parent[v]!;
        pathFlow = min(pathFlow, residualGraph[u]![v]!);
      }

      for (var v = sink; v != source; v = parent[v]!) {
        final u = parent[v]!;
        residualGraph[u]![v] = residualGraph[u]![v]! - pathFlow;
        residualGraph[v]!.putIfAbsent(u, () => 0);
        residualGraph[v]![u] = residualGraph[v]![u]! + pathFlow;


      }

      maxFlow += pathFlow;
    }

    return maxFlow;
  }
}



class Quadtree {
  final Rect bounds;
  final int capacity;
  final List<MapEntry<String, Offset>> points;
  Quadtree? northwest, northeast, southwest, southeast;
  bool divided = false;

  Quadtree({required this.bounds, this.capacity = 4}) : points = [];

  bool insert(String key, Offset point) {
    if (!bounds.contains(point)) {
```

```
      return false;
    }

    if (points.length < capacity) {
      points.add(MapEntry(key, point));
      return true;
    }

    if (!divided) {
      subdivide();
    }

    if (northwest!.insert(key, point)) return true;
    if (northeast!.insert(key, point)) return true;
    if (southwest!.insert(key, point)) return true;
    if (southeast!.insert(key, point)) return true;

    return false;
  }

  void subdivide() {
    final halfWidth = bounds.width / 2;
    final halfHeight = bounds.height / 2;
    final centerX = bounds.left + halfWidth;
    final centerY = bounds.top + half

Height;

    northwest = Quadtree(
        bounds: Rect.fromLTWH(bounds.left, bounds.top, halfWidth, halfHeight));
    northeast = Quadtree(
        bounds: Rect.fromLTWH(centerX, bounds.top, halfWidth, halfHeight));
    southwest = Quadtree(
        bounds: Rect.fromLTWH(bounds.left, centerY, halfWidth, halfHeight));
    southeast = Quadtree(
        bounds: Rect.fromLTWH(centerX, centerY, halfWidth, halfHeight));

    divided = true;
  }

  void query(Offset point, Function(String, Offset) callback) {
    if (!bounds.contains(point)) {
      return;
    }

    for (var entry in points) {
      callback(entry.key, entry.value);
    }

    if (divided) {
      northwest!.query(point, callback);
      northeast!.query(point, callback);
      southwest!.query(point, callback);
      southeast!.query(point, callback);
    }
  }
}




class MasterDetailLayoutAlgorithm extends LayoutAlgorithm {
```

```
  final double nodeWidth;
  final double nodeHeight;
  final double levelGap;

  MasterDetailLayo

utAlgorithm({
    this.nodeWidth = 200.0,
    this.nodeHeight = 100.0,
    this.levelGap = 50.0,
  });

  @override
  Map<String, Offset> calculateLayout(Domains domains, Size size) {
    final positions = <String, Offset>{};
    double currentX = levelGap;
    double currentY = levelGap;

    for (var domain in domains) {
      positions[domain.code] = Offset(currentX, currentY);

      for (var model in domain.models) {
        currentY += nodeHeight + levelGap;
        positions[model.code] = Offset(currentX, currentY);

        for (var entity in model.concepts) {
          currentY += nodeHeight + levelGap;
          positions[entity.code] =
              Offset(currentX + nodeWidth + levelGap, currentY);

          for (var child in entity.children) {
            currentY += nodeHeight + levelGap;
            positions[child.code] =
                Offset(currentX + 2 * (nodeWidth + levelGap), currentY);
          }
        }
      }
      currentX += 3 * (nodeWidth + levelGap);

      currentY = levelGap;
    }

    return positions;
  }
}




class GridLayoutAlgorithm extends LayoutAlgorithm {
  @override
  Map<String, Offset> calculateLayout(Domains domains, Size size) {
    final positions = <String, Offset>{};

    double x = 50;
    double y = 50;
    const double stepX = 200;
    const double stepY = 100;

    for (var domain in domains) {
      positions[domain.code] = Offset(x, y);
      y += stepY;
```

```
      for (var model in domain.models) {
        positions[model.code] = Offset(x + stepX, y);
        y += stepY;

        for (var entity in model.concepts) {
          positions[entity.code] = Offset(x + 2 * stepX, y);
          y += stepY;
        }

        y += stepY;
      }

      x += 3 * stepX;
      y = 50;
    }

    return positions;
  }
}
```

```
class ForceDirectedLayoutAlgorithm extends LayoutAlgorithm {
  final Map<String, Offset> positions = {};
  final Map<String, Offset> velocity = {};
  final double repulsionForce = 1000.0; // Adjusted re

pulsion force
  final double springForce = 0.1; // Spring force constant
  final int iterations = 500; // Reduced number of iterations for better
performance
  final double damping = 0.85; // Velocity damping factor to stabilize the
layout

  @override
  Map<String, Offset> calculateLayout(Domains domains, Size size) {
    final forces = <String, Offset>{};

    final random = Random();
    _initializePositions(domains, size, random);

    for (var i = 0; i < iterations; i++) {
      _applyForces(forces, size);
      _updatePositions(forces);
    }

    return positions;
  }

  void _initializePositions(Domains domains, Size size, Random random) {
    for (var domain in domains) {
      positions[domain.code] = Offset(random.nextDouble() * size.width,
random.nextDouble() * size.height);

      for (var model in domain.models) {
        positions[model.code] = Offset(random.nextDouble() * size.width,
random.nextDouble() * size.height);

        for (var entity in model.concepts) {
```

```
     positions[entity.code]    =    Offset(random.nextDouble()    *    size.width,
random.nextDouble() * size.height);

          for (var child in entity.children) {
            positions[child.code]  =  Offset(random.nextDouble()  *  size.width,
random.nextDouble() * size.height);
          }
        }
      }
    }
  }

  void _applyForces(Map<String, Offset> forces, Size size) {
    final   quadTree   =   Quadtree(bounds:   Rect.fromLTWH(0,   0,   size.width,
size.height));

    positions.forEach((key, position) {
      quadTree.insert(key, position);
    });

    forces.clear();

    positions.forEach((key, position) {
      var force = Offset.zero;

      quadTree.query(position, (otherKey, otherPosition) {
        if (key == otherKey) return;

        final direction = position - otherPosition;
        final distance = max(direction.distance, 0.1); // Avoid division by zero
        final repulsion = direction / distance * repulsionForce / (distance *
distance);

        force += repulsion;
      });


 forces[key] = force;
    });

    for (var domain in positions.keys) {
      for (var model in positions.keys) {
        if (domain == model) continue;
        final direction = positions[domain]! - positions[model]!;
        final distance = max(direction.distance, 1.0);
        final attraction = direction / distance * springForce * log(distance +
1);

        forces[domain] = (forces[domain] ?? Offset.zero) - attraction;
        forces[model] = (forces[model] ?? Offset.zero) + attraction;
      }
    }
  }

  void _updatePositions(Map<String, Offset> forces) {
    positions.forEach((key, position) {
      final force = forces[key] ?? Offset.zero;
      final   velocity   =   (this.velocity[key]   ??   Offset.zero)   +   force   *
springForce;

      positions[key] = position + velocity;
      this.velocity[key] = velocity * damping;
```

```
    });
  }
}



class DijkstraLayoutAlgorithm extends LayoutAlgorithm {
  @override
  Map<String, Offset> calculateLayout(Domains domains, Size size) {
    final pos

itions = <String, Offset>{};
    final graph = <String, Map<String, double>>{};

    for (var domain in domains) {
      positions[domain.code] = Offset(size.width / 2, size.height / 2);
      graph[domain.code] = {};

      for (var model in domain.models) {
        final modelPosition = Offset(size.width * 0.25, size.height * 0.25);
        positions[model.code] = modelPosition;
        graph[domain.code]![model.code] =
            _distance(positions[domain.code]!, modelPosition);

        for (var entity in model.concepts) {
          final entityPosition = Offset(size.width * 0.75, size.height * 0.75);
          positions[entity.code] = entityPosition;
          graph[model.code]![entity.code] =
              _distance(modelPosition, entityPosition);

          for (var child in entity.children) {
            final childPosition = Offset(size.width * 0.5, size.height * 0.5);
            positions[child.code] = childPosition;
            graph[entity.code]![child.code] =

      _distance(entityPosition, childPosition);
          }
        }
      }
    }

    final dijkstraPositions = _dijkstra(graph, domains.first.code, positions);
    return dijkstraPositions;
  }

  double _distance(Offset a, Offset b) {
    return (a - b).distance;
  }

  Map<String, Offset> _dijkstra(Map<String, Map<String, double>> graph,
      String start, Map<String, Offset> positions) {
    final distances = <String, double>{};
    final previous = <String, String?>{};
    final pq = SplayTreeMap<double, List<String>>();

    for (var node in graph.keys) {
      distances[node] = double.infinity;
      previous[node] = null;
      pq.putIfAbsent(double.infinity, () => []).add(node);
    }
```

```
    distances[start] = 0;
    pq.putIfAbsent(0, () => []).add(start);

    while (pq.isNotEmpty) {
      final u = pq[pq.firstKey()]!.removeAt(0);
      if (pq[pq.firstKey()]!.isEmpty) {
        pq.remove(pq.firstKey());
      }

      for (var neighbor in graph[u]!.keys) {
        final alt

 = distances[u]! + graph[u]![neighbor]!;
        if (alt < distances[neighbor]!) {
          pq[distances[neighbor]!]!.remove(neighbor);
          if (pq[distances[neighbor]!]!.isEmpty) {
            pq.remove(distances[neighbor]!);
          }
          distances[neighbor] = alt;
          previous[neighbor] = u;
          pq.putIfAbsent(alt, () => []).add(neighbor);
        }
      }
    }

    final dijkstraPositions = <String, Offset>{};
    for (var node in positions.keys) {
      dijkstraPositions[node] = positions[node]!;
    }

    return dijkstraPositions;
  }
}




class AVLTree {
  TreeNode? root;

  int height(TreeNode? node) {
    return node?.height ?? 0;
  }

  int max(int a, int b) {
    return (a > b) ? a : b;
  }

  TreeNode? rightRotate(TreeNode y) {
    if (y.left == null) return y; // Added null check
    TreeNode x = y.left!;
    TreeNode? T2 = x.right;

    x.right = y;
    y.left = T2;

    y.height = max(height(y.left), height(y.right)) + 1;
    x.height = max(

height(x.left), height(x.right)) + 1;

    return x;
```

```
  }

  TreeNode? leftRotate(TreeNode x) {
    if (x.right == null) return x; // Added null check
    TreeNode y = x.right!;
    TreeNode? T2 = y.left;

    y.left = x;
    x.right = T2;

    x.height = max(height(x.left), height(x.right)) + 1;
    y.height = max(height(y.left), height(y.right)) + 1;

    return y;
  }

  int getBalance(TreeNode? node) {
    if (node == null) return 0;
    return height(node.left) - height(node.right);
  }

  TreeNode insert(TreeNode? node, String key, Offset position) {
    if (node == null) return TreeNode(key, position);

    if (key.compareTo(node.key) < 0) {
      node.left = insert(node.left, key, position);
    } else if (key.compareTo(node.key) > 0) {
      node.right = insert(node.right, key, position);
    } else {
      return node;
    }

    node.height = max(height(node.left), height(node.right)) + 1;

    int balance = getBalance(node);

    if (balance > 1 && key.compareTo(node.left!.k

ey) < 0) {
      return rightRotate(node)!;
    }

    if (balance < -1 && key.compareTo(node.right!.key) > 0) {
      return leftRotate(node)!;
    }

    if (balance > 1 && key.compareTo(node.left!.key) > 0) {
      node.left = leftRotate(node.left!)!;
      return rightRotate(node)!;
    }

    if (balance < -1 && key.compareTo(node.right!.key) < 0) {
      node.right = rightRotate(node.right!)!;
      return leftRotate(node)!;
    }

    return node;
  }

  Offset? search(TreeNode? node, String key) {
    if (node == null) return null;
```

```
    if (key == node.key) return node.position;

    if (key.compareTo(node.key) < 0) {
      return search(node.left, key);
    } else {
      return search(node.right, key);
    }
  }

  void insertNode(String key, Offset position) {
    root = insert(root, key, position);
  }

  Offset? getNodePosition(String key) {
    return search(root, key);
  }
}




class MSTLayoutAlgorithm extends LayoutAlgorithm {
  @override
  Map<String, Offset> calculat

eLayout(Domains domains, Size size) {
    final positions = <String, Offset>{};
    final edges = <Edge>[];

    for (var domain in domains) {
      final domainPosition = Offset(size.width / 2, size.height / 2);
      positions[domain.code] = domainPosition;

      for (var model in domain.models) {
        final modelPosition = Offset(size.width * 0.25, size.height * 0.25);
        positions[model.code] = modelPosition;
        edges.add(Edge(
            domain.code, model.code, _distance(domainPosition, modelPosition)));

        for (var entity in model.concepts) {
          final entityPosition = Offset(size.width * 0.75, size.height * 0.75);
          positions[entity.code] = entityPosition;
          edges.add(Edge(model.code, entity.code,
              _distance(modelPosition, entityPosition)));

          for (var child in entity.children) {
            final childPosition = Offset(size.width * 0.5, size.height * 0.5);
            positions[child.code] = childPosition;

      edges.add(Edge(entity.code, child.code,
                _distance(entityPosition, childPosition)));
          }
        }
      }
    }

    final mst = _kruskalMST(edges, positions);
    return mst;
  }

  double _distance(Offset a, Offset b) {
    return (a - b).distance;
```

```
  }

  Map<String, Offset> _kruskalMST(
      List<Edge> edges, Map<String, Offset> positions) {
    edges.sort((a, b) => a.weight.compareTo(b.weight));
    final parent = <String, String>{};
    final rank = <String, int>{};

    String find(String u) {
      if (parent[u] != u) {
        parent[u] = find(parent[u]!);
      }
      return parent[u]!;
    }

    void union(String u, String v) {
      final rootU = find(u);
      final rootV = find(v);
      if (rootU != rootV) {
        if (rank[rootU]! > rank[rootV]!) {
          parent[rootV] = rootU;
        } else if (rank[rootU]! < rank[rootV]!) {
          parent[rootU] = rootV;
        } else {
          parent[rootV] = rootU;
          rank[rootU] = ra

nk[rootU]! + 1;
        }
      }
    }

    for (var key in positions.keys) {
      parent[key] = key;
      rank[key] = 0;
    }

    final mst = <Edge>[];
    for (var edge in edges) {
      if (find(edge.u) != find(edge.v)) {
        mst.add(edge);
        union(edge.u, edge.v);
      }
    }

    final mstPositions = <String, Offset>{};
    for (var edge in mst) {
      mstPositions[edge.u] = positions[edge.u]!;
      mstPositions[edge.v] = positions[edge.v]!;
    }

    return mstPositions;
  }
}




abstract class Component {
  void update(double dt);

  void render(Canvas canvas);
```

```
}

class Artefact extends PositionComponent {
  final String label;
  final double width;
  final double height;
  final Paint _paint;

  Artefact(
    super.position, {
    required this.label,
    required this.width,
    required this.height,
    Color color = Colors.blue,
  }) : _paint = Paint()..color = color;

  @override
  void render(Canvas canvas) {
    super.render(canvas);
    canvas.drawRe

ct(Rect.fromLTWH(0, 0, width, height), _paint);
    TextPainter tp = TextPainter(
      text: TextSpan(
        text: label,
        style: TextStyle(color: Colors.white, fontSize: 14),
      ),
      textAlign: TextAlign.center,
      textDirection: TextDirection.ltr,
    );
    tp.layout(maxWidth: width);
    tp.paint(canvas, Offset((width - tp.width) / 2, (height - tp.height) / 2));
  }
}

class PositionComponent extends Component {
  Offset position;

  PositionComponent(this.position);

  @override
  void update(double dt) {}

  @override
  void render(Canvas canvas) {
    canvas.drawCircle(
      position,
      100,
      Paint()
        ..color = Colors.orange
        ..blendMode = BlendMode.colorBurn,
    );
  }
}

class RenderComponent extends Component {
  final Paint paint;
  final Rect rect;
  final double glow;

  RenderComponent(this.paint, this.rect, {this.glow = 0.0});
```

```
  @override
  void update(double dt) {}

  @override
  void render(Canvas canvas) {
    if (glow > 0

.0) {
      final glowPaint = Paint()
        ..color = paint.color.withOpacity(0.5)
        ..maskFilter = MaskFilter.blur(BlurStyle.normal, glow);
      canvas.drawRect(rect.inflate(glow), glowPaint);
    }
    canvas.drawRect(rect, paint);
  }
}

class LineComponent extends Component {
  final Offset start;
  final Offset end;
  final Paint paint;
  final String fromToName;
  final String toFromName;
  final TextStyle fromTextStyle;
  final TextStyle toTextStyle;
  final double margin;

  LineComponent({
    required this.start,
    required this.end,
    required this.fromToName,
    required this.toFromName,
    Color color = Colors.black,
    required this.fromTextStyle,
    required this.toTextStyle,
    this.margin = 100.0,
  }) : paint = Paint()
          ..color = color
          ..strokeWidth = 1;

  @override
  void update(double dt) {}

  @override
  void render(Canvas canvas) {
    canvas.drawLine(start, end, paint);

    final direction = (end - start).direction;

    _d

rawText(canvas, fromToName, start, fromTextStyle, direction, margin);

    _drawText(canvas, toFromName, end, toTextStyle, direction + pi, margin);
  }

  void _drawText(Canvas canvas, String text, Offset position, TextStyle style,
      double direction, double offset) {
    final textPainter = TextPainter(
      text: TextSpan(
        text: text,
        style: style,
```

```
      ),
      textDirection: TextDirection.ltr,
    );

    textPainter.layout();

    final dx = cos(direction) * offset;
    final dy = sin(direction) * offset;

    final adjustedPosition = position + Offset(dx, dy);

    textPainter.paint(
        canvas,
        Offset(adjustedPosition.dx - textPainter.width / 2,
            adjustedPosition.dy - textPainter.height / 2));
  }
}

class TextComponent extends Component {
  final String text;
  final Offset position;
  final TextStyle style;
  final double padding;
  final Color backgroundColor;

  TextComponent({
    required this.text,
    required this.position,

   required this.style,
    this.padding = 4.0,
    this.backgroundColor = Colors.black,
  });

  @override
  void update(double dt) {}

  @override
  void render(Canvas canvas) {
    final textSpan = TextSpan(text: text, style: style);
    final textPainter = TextPainter(
      text: textSpan,
      textAlign: TextAlign.center,
      textDirection: TextDirection.ltr,
    );
    textPainter.layout();

    final backgroundRect = Rect.fromLTWH(
      position.dx - textPainter.width / 2 - padding,
      position.dy - textPainter.height / 2 - padding,
      textPainter.width + padding * 2,
      textPainter.height + padding * 2,
    );

    final paint = Paint()..color = backgroundColor;

    canvas.drawRect(backgroundRect, paint);
    textPainter.paint(
        canvas,
        Offset(position.dx - textPainter.width / 2,
            position.dy - textPainter.height / 2));
  }
```

```
}



class System {
  final List<Node> nodes = [];

  void addNode(Node node) {
    nodes.add(node);
  }

  void r

ender(Canvas canvas) {
    for (var node in nodes) {
      for (var component in node.components
          .where((component) => component is! TextComponent)) {
        component.render(canvas);
      }
    }
  }

  void renderText(Canvas canvas) {
    for (var node in nodes) {
      for (var component in node.components.whereType<TextComponent>()) {
        component.render(canvas);
      }
    }
  }

  void update(double dt) {
    for (var node in nodes) {
      node.update(dt);
    }
  }
}




class Node {
  final List<Component> components = [];

  void addComponent(Component component) {
    components.add(component);
  }

  void update(double dt) {
    for (var component in components) {
      component.update(dt);
    }
  }

  void render(Canvas canvas) {
    for (var component in components) {
      component.render(canvas);
    }
  }
}
```

```
enum NodeType {
  domain,
  model,
  entity,
}




class DraggableArtefact extends StatefulWidget {
  final Artefact artefact;

  DraggableAr

tefact({required this.artefact});

  @override
  _DraggableArtefactState createState() => _DraggableArtefactState();
}

class _DraggableArtefactState extends State<DraggableArtefact> {
  Offset position = Offset(100, 100); // Initial position

  @override
  Widget build(BuildContext context) {
    return Positioned(
      left: position.dx,
      top: position.dy,
      child: Draggable<Artefact>(
        data: widget.artefact,
        feedback: Material(
          color: Colors.transparent,
          child: _buildArtefactWidget(widget.artefact),
        ),
        child: _buildArtefactWidget(widget.artefact),
        childWhenDragging: Container(),
        onDragEnd: (dragDetails) {
          setState(() {
            position = dragDetails.offset;
          });
        },
      ),
    );
  }

  Widget _buildArtefactWidget(Artefact artefact) {
    return Container(
      width: 100,
      height: 100,
      color: Colors.blue,
      child: Center(
        child: Text(
          artefa

ct.label,
          style: TextStyle(color: Colors.white),
        ),
      ),
    );
  }
}
```

```
class LayoutAlgorithmIcon extends StatefulWidget {
  final IconData icon;
  final String name;
  final VoidCallback onTap;
  final bool isActive;

  const LayoutAlgorithmIcon({
    super.key,
    required this.icon,
    required this.name,
    required this.onTap,
    required this.isActive,
  });

  @override
  _LayoutAlgorithmIconState createState() => _LayoutAlgorithmIconState();
}

class _LayoutAlgorithmIconState extends State<LayoutAlgorithmIcon> {
  bool _isHovering = false;

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: widget.onTap,
      child: Tooltip(
        message: widget.name,
        child: Padding(
          padding: const EdgeInsets.all(8.0), // Add padding around the icon
          child: MouseRegion(
            onEnter: (_) => setState(() => _isHovering = true),
            onExit: (_) => setState(() => _isHovering = false),

           child: AnimatedContainer(
              duration: Duration(milliseconds: 200),
              child: Column(
                mainAxisSize: MainAxisSize.min,
                children: [
                  Icon(
                    widget.icon,
                    size: 17,
                    color: widget.isActive || _isHovering
                        ? Theme.of(context).colorScheme.primary
                        : Theme.of(context).colorScheme.secondary,
                  ),
                  SizedBox(height: 4.0), // Add spacing between icon and text
                ],
              ),
            ),
          ),
        ),
      ),
    );
  }
}



class TreeNode {
  String key;
  Offset position;
```

```
  TreeNode? left;
  TreeNode? right;
  int height;
  List<TreeNode> children;

  TreeNode(this.key, this.position)
      : height = 1,
        children = [];
}


class Edge {
  final String u;
  final String v;
  final double weight;

  Edge(this.u, this.v, this.weight);
}




cl

ass GameLoop {
  final System system;
  final AnimationManager animationManager;
  final double updateInterval;
  late Timer _timer;

  GameLoop({
    required this.system,
    required this.animationManager,
    this.updateInterval = 1 / 60, // 60 FPS
  });

  void start() {
    _timer = Timer.periodic(
        Duration(milliseconds: (updateInterval * 1000).round()), _update);
  }

  void _update(Timer timer) {
    double dt = updateInterval;
    animationManager.update(dt);
    system.update(dt);
    system
        .render(Canvas(PictureRecorder())); // Replace with your rendering logic
  }

  void stop() {
    _timer.cancel();
  }
}



class AnimationManager {
  final List<Animation> animations = [];

  void addAnimation(Animation animation) {
    animations.add(animation);
  }
```

```
  void update(double dt) {
    for (var animation in List.from(animations)) {
      animation.update(dt);
      if (animation.elapsedTime >= animation.duration) {
        animations.remove(animation);

}
    }
  }
}


class Animation {
  final double duration;
  double elapsedTime = 0;
  final void Function(double progress) onUpdate;
  final void Function() onComplete;

  Animation({
    required this.duration,
    required this.onUpdate,
    required this.onComplete,
  });

  void update(double dt) {
    elapsedTime += dt;
    double progress = (elapsedTime / duration).clamp(0.0, 1.0);
    onUpdate(progress);
    if (elapsedTime >= duration) {
      onComplete();
    }
  }
}



abstract class IDomainLayout {
  Domain get domain;

  Model get model;

  Widget get focus;

  Widget get secondary;

  Widget get auxiliary;

  Widget build(BuildContext context);

  void onDomainSelected(Domain domain);

  void onModelSelected(Model model);

  void onEntitySelected(Entity entity);
}
```