

# Material Complementar

## Prevenção de Anomalias Arquiteturais na Otimização de Projeto de Linha de Produto

Tiago Tadeu Madrigar<sup>1</sup>, Thelma Elita Colanzi<sup>1</sup>, Willian Oizumi<sup>2</sup>, and Alessandro Garcia<sup>2</sup>

<sup>1</sup> UEM - Universidade Estadual do Maringá - Maringá, PR, Brasil

[tiago@madrigar.com.br](mailto:tiago@madrigar.com.br), [thelma@din.uem.br](mailto:thelma@din.uem.br)

<sup>2</sup> PUC-Rio - Rio de Janeiro, RJ, Brasil

[woizumi@inf.puc-rio.br](mailto:woizumi@inf.puc-rio.br), [afgarcia@inf.puc-rio.br](mailto:afgarcia@inf.puc-rio.br)

### 1 Introdução

Neste documento são exibidos artefatos e resultados que, por falta de espaço, tiveram que ser omitidos ou parcialmente apresentados no artigo *Prevenção de Anomalias Arquiteturais na Otimização de Projeto de PLA* publicado no CIBSE2020.

### 2 Anomalias Arquiteturais identificadas no estudo de Perissato *et al* (2018)

Em nosso trabalho prévio [3], fizemos um levantamento sobre uma série de anomalias arquiteturais. Esse levantamento teve como objetivo identificar quais anomalias arquiteturais existiam nas instâncias originais dos projetos de PLA (*Product Line Architecture*) utilizados nos experimentos já realizados, bem como os tipos de anomalias presentes nas soluções geradas pela ferramenta OPLA-Tool. Ao todo foram encontrados 8 tipos de anomalias arquiteturais. A Tabela 1 apresenta uma descrição completa de cada uma dessas anomalias, juntamente com seu método de identificação e a diretriz proposta em [3] para resolver as anomalias: *Link Overload*, *Concern Overload*, *Unused Interface*, *Unused Brick*, *Sloppy Delegation*, *Dependency Cycle*, *Connector Envy* e *Large Class*.

### 3 OPLA-Tool-ASP

A principal contribuição deste trabalho é a implementação das diretrizes propostas em nosso estudo prévio [3] para as três anomalias mais frequentes nas soluções geradas pela OPLA-Tool. Essa implementação gerou a primeira versão da OPLA-Tool-ASP (uma evolução da ferramenta OPLA-Tool), que além de otimizar projetos de PLA, detecta e previne as anomalias *Unused Interface*, *Unused Brick* e *Concern Overload*.

### 3.1 Prevenção das anomalias *Unused Interface* e *Unused Brick*

A anomalia *Unused Brick* é gerada com a existência da *Unused Interface*, então, em tese, resolvendo a primeira anomalia, a segunda será resolvida automaticamente. A Figura 1 apresenta um fluxograma do algoritmo NSGA-II juntamente com a solução utilizada para resolução da anomalia *Unused Interface* e *Unused Brick*.

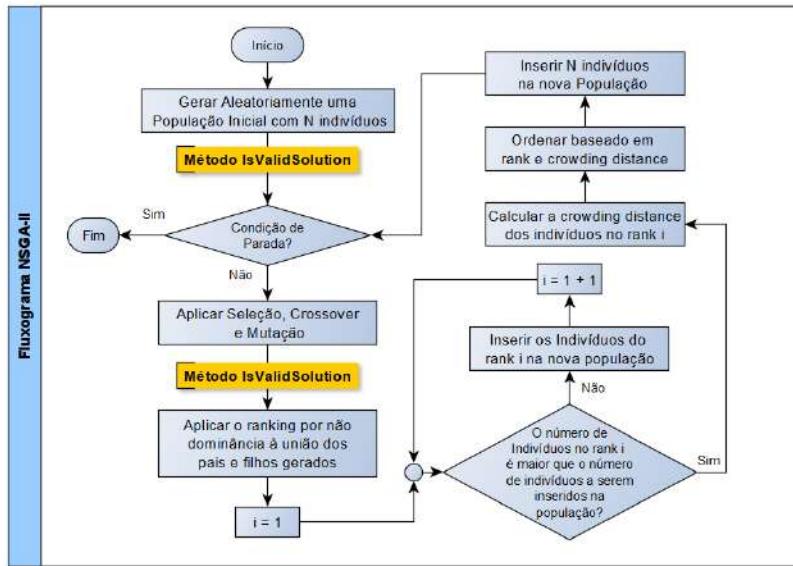


Figura 1: Fluxograma NSGA-II e Resolução da anomalia *Unused Interface* e *Unused Brick*

A primeira mudança acontece no momento da criação da população inicial, pois se a população inicial for criada com interfaces vazias e componentes desconexos, depois de acontecer os cruzamentos e mutações, a população filha também apresentará o problema. Na sequência, é realizada uma alteração na criação da população filha. Em ambas alterações ocorreu a inserção do método *isValidSolution*, cuja função é descartar as soluções que contêm classes e interfaces desconexas, retornando apenas soluções livres da anomalia. O método *isValidSolution* é apresentado no Algoritmo 1.

O método *isValidSolution* é executado sempre que uma nova solução é criada. Ele verifica se as interfaces da solução não são vazias e estão conectadas no projeto, retornando false quando uma interface isolada/vazia for identificada (linha 6). Quando este método retorna false, a solução é descartada evitando que ela seja inserida na população que seguirá no processo de busca. Somente as soluções cujo retorno de *isValidSolution* é true são adicionadas à população corrente.

**Algorithm 1:** MÉTODO ISVALIDSOLUTION

---

```

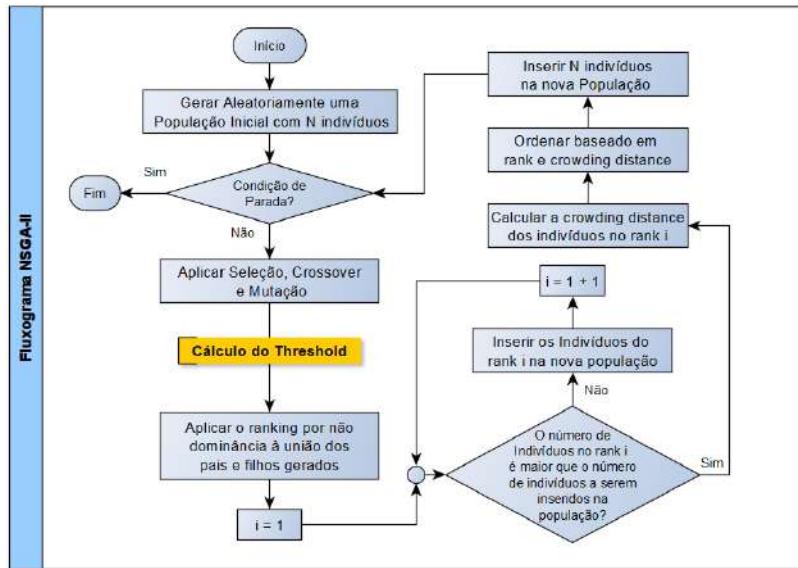
1 public boolean isValidSolution(Architecture solution)
2 final List<Interface> allInterfaces = new
   ArrayList<Interface>(solution.getAllInterfaces());
3 if (!allInterfaces.isEmpty()) then
4   for (Interface itf : allInterfaces) do
5     if ((itf.getImplementors().isEmpty()) and (itf.getDependents().isEmpty()) and
       (itf.getOperations().isEmpty())) then
6       | return false;
7     end
8   end
9 end
10 return true;

```

---

**3.2 Prevenção da anomalia *Concern Overload***

A Figura 2 apresenta o primeiro passo para resolução da anomalia *Concern overload*, essa alteração acontece no Algoritmo NSGA-II, após a criação da população inicial e população filha é realizado o cálculo do *threshold*. Para que uma classe ou interface seja anômala todos as características (do inglês *concerns*) devem ser contadas, calculando, assim, a média e somando ao desvio padrão. O resultado deste cálculo é denominado de *threshold* e foi proposto por [2]. Uma vez excedido o *threshold* a classe ou a interface é considerada anômala.

Figura 2: Resolução da anomalia *Concern Overload*

O Algoritmo 2 apresenta o método de detecção da classe ou da interface que excedeu o *threshold*, explicado anteriormente. Essa alteração acontece logo após a criação da população filha do algoritmo NSGA-II.

Incialmente no algoritmo são listados todos os pacotes da solução original e então é realizada a listagem de cada pacote da solução (linha 5) e, para cada pacote listado, são ordenadas todas suas classes. Na sequência, conta-se a quantidade de *concerns* da classe e o valor é salvo na lista criada. O processo é repetido para as interfaces (linha 12). Na sequência é realizado o cálculo da média e do desvio padrão respectivamente, a final somamos média juntamente com desvio padrão, assim encontramos o *threshold*. O valor obtido fracionado é arredondamento para cima.

---

**Algorithm 2:** DETECÇÃO DA ANOMALIA CONCERN OVERLOAD

---

```

1 public boolean detectCO(Solution solution) throws JMException {
2 final Architecture arch = ((Architecture) solution.getDecisionVariables()[0]);
3 final List < Package > allPackage = new ArrayList < Package > (arch.getAllPackages());
4 if (!allPackage.isEmpty()) {
5     for (Package selectedPackage: allPackage) {
6         List < Class > lstClass = new ArrayList < >(selectedPackage.getAllClasses());
7         for (Class selectedClass: lstClass) {
8             /List < Concern > lstConcern = new ArrayList <
9                 >(selectedClass.getOwnConcerns());
10            if (lstConcern.size() > threshold) then
11                | return true;
12            end
13            List < Interface > lstInterface = new ArrayList <
14                 >(selectedPackage.getAllInterfaces());
15            for (Class selectedClass: lstClass) {
16                List < Concern > lstConcern = new ArrayList <
17                     >(selectedInterface.getOwnConcerns());
18                if (lstConcern.size() > threshold) then
19                    | return true;
20                end
21            end
22        end
23    end
24}

```

---

Uma vez identificada a anomalia, a diretriz proposta em [3] sugere a aplicação do operador de busca *Feature-Driven Operator*, que consegue modularizar características e, como consequência, pode diminuir a ocorrência da anomalia *Concern Overload*. O Algoritmo 3 apresenta a aplicação do operador de busca (linha 7) e sua aplicação ocorre na classe PLAFeatureMutation, responsável pelos operadores de mutação.

## 4 Projeto do Estudo Experimental

As anomalias arquiteturais *Concern Overload* e *Link Overload* avaliadas neste trabalho precisaram ter um *threshold* definido para identificar sua ocorrência.

**Algorithm 3:** APLICAÇÃO DO FEATURE DRIVEN OPERATOR

---

```

1 public void doMutation(double probability, Solution solution) throws Exception {
2     String scope = "sameComponent";
3     String scopeLevels = "allLevels";
4     String selectedOperator = ;
5     if (detectCO(solution)) then
6         System.out.println("Existe CO -> Aplicar Feature-Driven Operator");
7         selectedOperator = "featureMutation";
8     else
9         int r = PseudoRandom.nextInt(0, this.mutationOperators.size() - 1);
10        HashMap<Integer, String> operatorMap = new HashMap<>();
11    end
12 end

```

---

Para a anomalia *Concern Overload* seguiu-se a fórmula indicada por Garcia [2] e foram contadas as características associadas ao nome da classe ou interface. Devido à formula resultar em valores fracionários, os valores foram arredondados para cima. A fórmula proposta por Garcia pode ser formalizada:

$$|\{z_j \bullet (j \in R) \wedge (P(z_j|b) > th_{zb})\}| > th_{co}$$

Onde,  $0 \leq th_{zb} \leq 1$  é o limite que indica que a representação de um concern; e  $th_{co} \in \mathbb{R}$  é um limite que indica o número máximo aceitável de *concerns* por classes ou interfaces.

Para a anomalia *Link Overload* foi necessário realizar uma inspeção em cada classe e interface em busca dos relacionamentos e suas direcionalidades. E para descobrir quais eram links de entrada, saída e ambos foi necessário uma verificação em relação a seus relacionamentos. O Relacionamento de Dependência é um relacionamento no qual um elemento, o cliente depende de outro elemento, o fornecedor. O relacionamento de realização é um relacionamento entre dois elementos, nos quais um elemento (o cliente) realiza o comportamento que o outro elemento (o fornecedor) especifica. O relacionamento de abstração geralmente é definido como um relacionamento entre cliente (s) e fornecedor (es) em que o cliente (subconjunto de origem) depende do fornecedor (subconjunto de destino). O relacionamento de associação descreve um vínculo entre classes. Através da multiplicidade esse relacionamento determina as instâncias que uma classe estão de alguma forma ligadas às instâncias da outra classe. Para todos os casos para localizar a navegação e descobrir quem é o cliente e o fornecedor foram considerados links de entrada para os clientes que possuíam identificador igual ao relacionamento. Os links sem direção foram considerados bi-direcionais. Uma vez contados todos os relacionamentos de cada classe e interface seguiu-se a fórmula indicada por Garcia [2] para o cômputo do *threshold* para cada direcionalidade. Os valores fracionados obtidos foram arredondados para cima. A Tabela 2 apresenta os valores obtidos para cada LPS, sendo que o rótulo ambos é utilizado para representar os relacionamentos bi-direcionais.

## 5 Resultados

A Tabela 3 apresenta o valor de *fitness* de cada solução obtida pelas ferramentas OPLA-Tool e OPLA-Tool-ASP, isto é, o valor de avaliação de cada uma das funções objetivo que foram utilizadas (COE, ACLASS e FM).

Também é apresentada a porcentagem de melhoria do fitness de cada solução em relação ao *fitness* da PLA original dada como entrada para ferramenta, assim foi possível analisar qual solução obteve melhora ou piora em determinado aspecto. O valor de *fitness* original da PLA AGM é (COE=30, ACLASS=26, FM=758) e da PLA MM é (COE=14, ACLASS=28, FM=1122).

No repositório do GitHub<sup>3</sup> estão sendo disponibilizados os arquivos das soluções de projeto de PLA obtidos pelas duas ferramentas na pasta denominada **Solucoes Obtidas**. Além disso, estão sendo disponibilizadas três planilhas do *Microsoft®Office Excel* com informações complementares conforme segue:

- **OPLA-Tool.xlsx**: dados sobre as anomalias das soluções obtidas pela ferramenta OPLA-Tool;
- **OPLA-Tool-ASP.xlsx**: dados sobre as anomalias das soluções obtidas pela ferramenta OPLA-Tool-ASP;
- **Totais.xlsx**: apresenta os totais e análises feitas comparando as duas ferramentas.

## 6 Anexos (Soluções encontradas)

A seguir são apresentadas todas as 36 soluções não-dominadas geradas pelas duas ferramentas após as 30 execuções dos experimentos na seguinte ordem: soluções da OPLA-Tool para a AGM e para a MM e soluções da OPLA-Tool-ASP para a AGM e para a MM.

## Referências

1. Fowler, M., Beck, K.: Refactoring: improving the design of existing code. Addison-Wesley Professional (1999)
2. Garcia, J.: A unified framework for studying architectural decay of software systems. University of Southern California (2014)
3. Perissato, E.G., Neto, J.C., Colanzi, T.E., Oizumi, W., Garcia, A.: On identifying architectural smells in search-based product line designs. In: VII Brazilian Symposium on Software Components, Architectures, and Reuse. pp. 13–22 (2018)

---

<sup>3</sup> <https://github.com/otimizes/opla-tool-asp/>

Tabela 1: Anomalias arquiteturais identificadas em Perissato *et al* (2018)

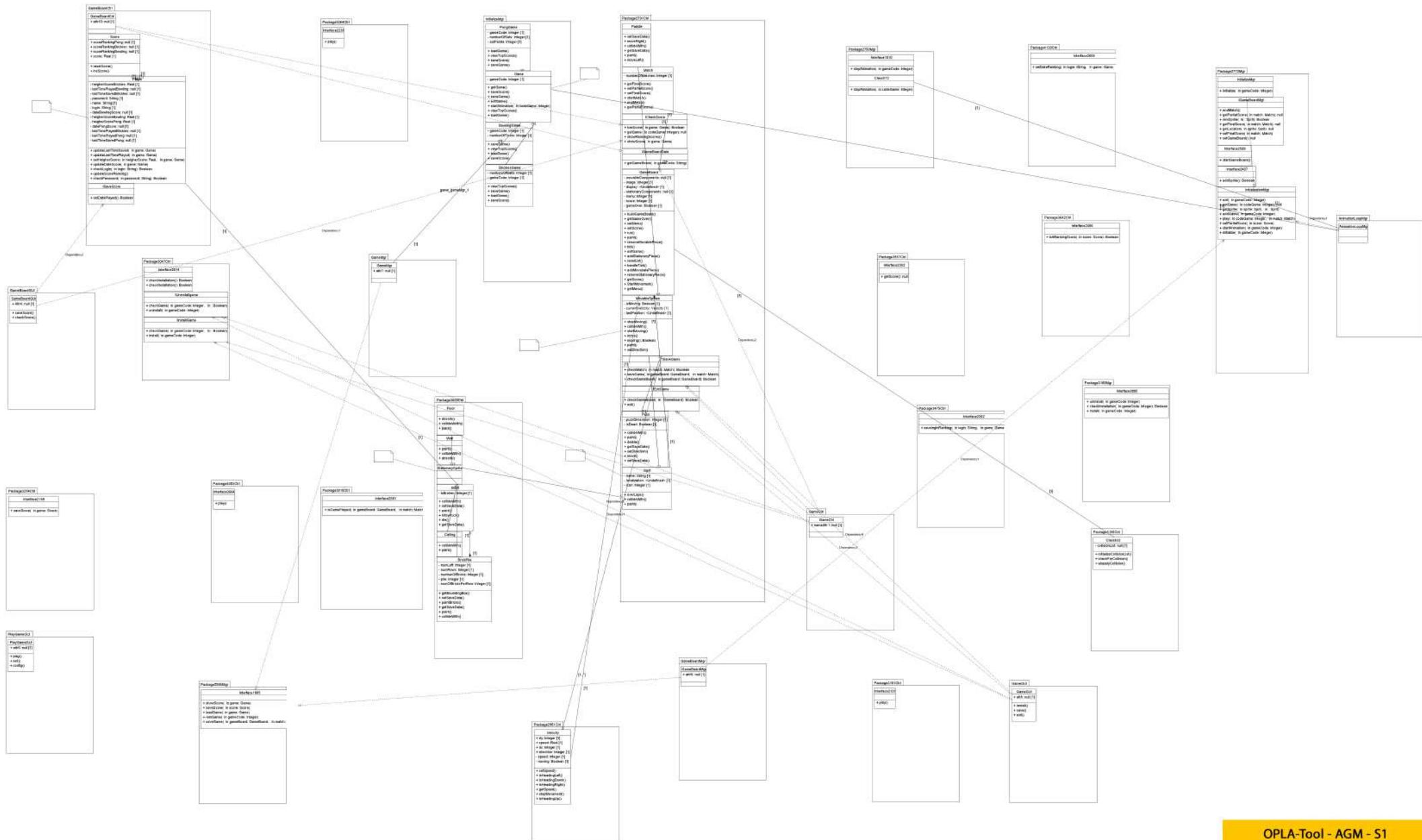
Tipo	Definição
Link Overload	<p><b>Descrição:</b> Um componente que possui excessiva dependência de outros componentes, prejudicando assim a separação das funcionalidades e o isolamento das mudanças. Essas dependências podem se manifestar como links de entrada, de saída, ou ambos [2].</p> <p><b>Identificação:</b> o algoritmo proposto por Garcia [2] foi utilizado para definir o valor limite para cada tipo de link. Relacionamentos de dependência, realização, herança, etc. foram considerados links, bem como uma classe possuir um atributo que é do tipo de outra classe. O valor limite foi calculado para cada direcionalidade: entrada, saída e bidirecional (vide Tabela 2), arredondando valores fracionários. Depois, fez-se a contagem em todas classes e interfaces.</p> <p><b>Diretriz:</b> Penalizar o <i>fitness</i> da solução, de modo que ela seja pior avaliada no processo de seleção dos melhores projetos para permanecerem no processo de otimização.</p>
Concern Overload	<p><b>Descrição:</b> Um componente que possui um número excessivo de características, quebrando o princípio da responsabilidade única [2].</p> <p><b>Identificação:</b> as características de cada LPS foram consideradas como sendo os <i>concerns</i> do projeto de ALP. O algoritmo proposto por Garcia [2] foi utilizado contando as características associadas às classes e interfaces do projeto. A Tabela 2 apresenta os valores limite obtidos para cada LPS. Valores fracionários foram arredondados para cima.</p> <p><b>Diretriz:</b> Sugere-se aplicação do operador de busca, <i>Feature-Driven Operator</i>, para modularizar alguma das características associadas à classe/interface anômala</p>
Unused Interface	<p><b>Descrição:</b> Uma interface de componente é dita não utilizada quando não está ligada a nenhum componente, adicionando, portanto, uma complexidade desnecessária ao sistema e dificultando sua manutenção [2].</p> <p><b>Identificação:</b> foram consideradas para esta anomalia as interfaces e classes isoladas, ou seja, que não possuíam nenhum relacionamento.</p> <p><b>Diretriz:</b> Sugere-se considerar inválida a solução que contenha essas anomalias e então descartá-la.</p>
Unused Brick	<p><b>Descrição:</b> Ocorre quando todas as interfaces de um componente apresentam a anomalia <i>Unused Interface</i>, ou seja, quando nenhuma de suas interfaces possui ligação a outro componente [2].</p> <p><b>Identificação:</b> foram considerados os pacotes em que nenhuma interface ou classe possuía relacionamentos.</p> <p><b>Diretriz:</b> Sugere-se considerar inválida a solução que contenha essas anomalias e então descartá-la.</p>
Sloppy Delegation	<p><b>Descrição:</b> Um componente que delega uma quantidade muito pequena de funcionalidades a outros componentes, que ele próprio poderia realizar [2]. A autossuficiência na realização de uma funcionalidade pode ser verificada quando todos os dados necessários a ela fazem parte do estado desse componente.</p> <p><b>Identificação:</b> foi realizada uma avaliação a fim de encontrar componentes (classes ou interfaces) que delegavam uma quantidade muito pequena de funcionalidades a outros componentes. Na avaliação foi considerada delegação de funcionalidade quando um componente que contém todos os dados necessários para realizar essa funcionalidade não a realiza, mas sim a repassa a um outro componente. Foi computada a quantidade de métodos delegados indevidamente.</p> <p><b>Diretriz:</b> Sugere-se que a solução tenha seu <i>fitness</i> penalizado, já que a aplicação de operadores de busca não garantiria a coesão do método com a classe que possui os dados necessários a ele.</p>
Dependency Cycle	<p><b>Descrição:</b> Um conjunto de componentes que se liga de forma que um dependa do outro, formando uma cadeia circular, e prejudicando a manutenção dos mesmos. Desse modo, mudanças em um desses componentes possivelmente afetam todos os outros componentes do ciclo [2].</p> <p><b>Identificação:</b> considerou-se os ciclos de dependência entre classes e/ou interfaces, ou seja, classes e/ou interfaces com ligação mútua.</p> <p><b>Diretriz:</b> Sugere-se penalizar o <i>fitness</i> da solução avaliada.</p>
Connector Envy	<p><b>Descrição:</b> Um componente que apresenta uma funcionalidade com elevado nível de interação, que deveria ser delegada a um conector, prejudicando assim o entendimento da funcionalidade e o reuso [2].</p> <p><b>Identificação:</b> Como os projetos de ALP analisados neste estudo não contém informações que permitam a diferenciação sistemática entre componentes e conectores, foi realizada uma avaliação interpretativa para identificar componentes que apresentassem alguma funcionalidade com elevado nível de interação.</p> <p><b>Diretriz:</b> Sugere-se penalizar o <i>fitness</i> da solução avaliada, de modo que ela não seja priorizada pelo algoritmo.</p>
Large Class	<p><b>Descrição:</b> Uma classe que possui quantidade excessiva de atributos ou métodos, indicando normalmente que a mesma possui um excesso de responsabilidades [2].</p> <p><b>Identificação:</b> foram consideradas <i>large classes</i> as classes ou interfaces que possuíam mais de 10 atributos, ou mais de 10 operações, ou mais de 15 elementos somando atributos e operações. Esses valores foram definidos considerando o tamanho médio das classes dos projetos originais das LPSs.</p> <p><b>Diretriz:</b> Sugere-se aplicar operadores de busca que movam atributos ou métodos para outras classes, de modo a reduzir o tamanho da classe anômala.</p>

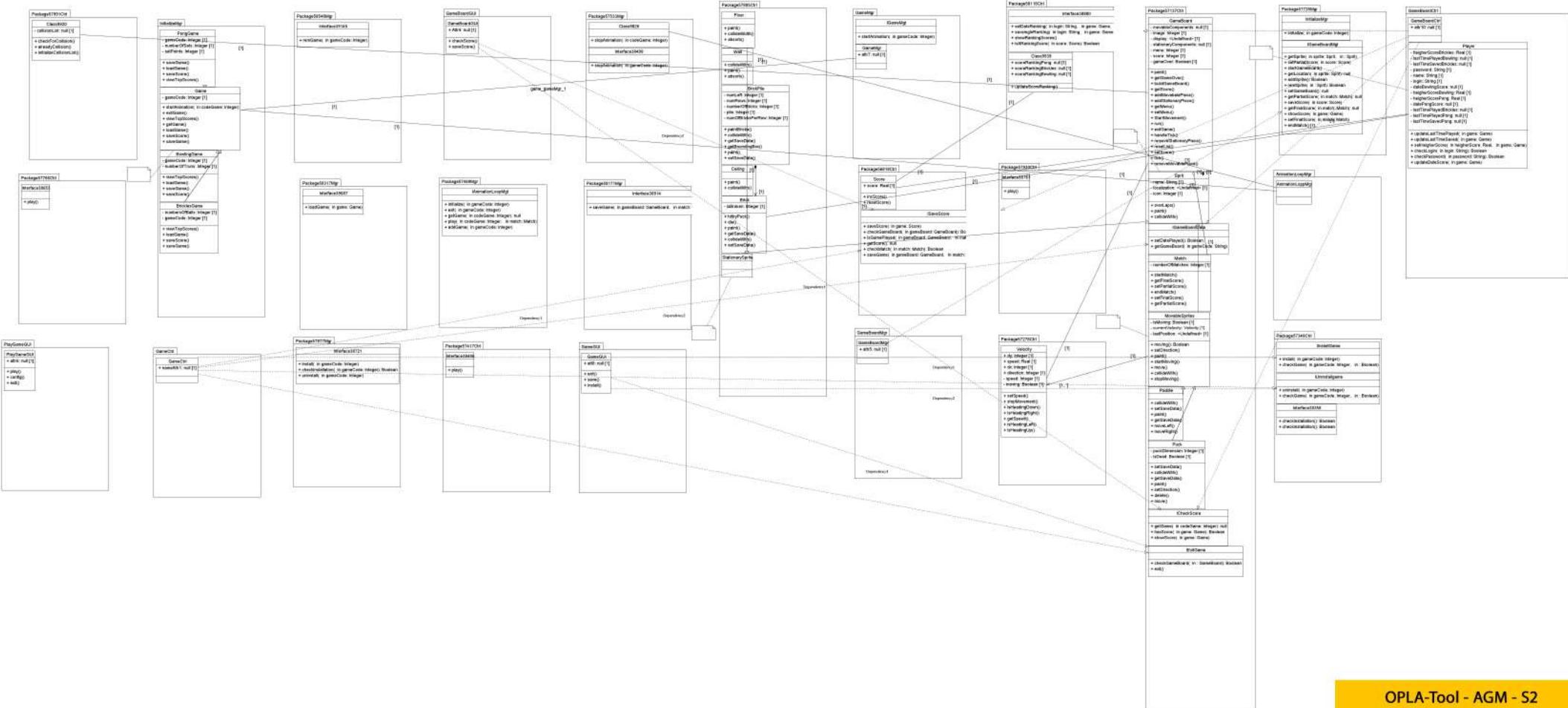
Tabela 2: Thresholds utilizados

Threshold	AGM	MM
Concern Overload	4	5
Link Overload (entrada)	2	2
Link Overload (saída)	2	3
Link Overload (ambos)	1	0

Tabela 3: Valor de Fitness e Porcentagem

PLA	OPLA-Tool		OPLA-Tool-ASP		PLA	OPLA-Tool		OPLA-Tool-ASP	
	Soluções	COE - ACCLASS - FM	COE - ACCLASS - FM	COE - ACCLASS - FM		COE - ACCLASS - FM	COE - ACCLASS - FM	COE - ACCLASS - FM	COE - ACCLASS - FM
AGM	Sol1	(46,10,525) -53.3%,61.5%,30.7%	(27,26,743) 10.0%,0.0%,2.0%		Sol1	(60,10,806) -328.6%,64.3%,28.2%	(28,7,1075) -100.0%,75.0%,4.2%		
	Sol2	(39,9,531) -30.0%,65.4%,29.9%	(16,29,711) 46.7%,-11.5%,6.2%		Sol2	(47,2,866) -235.7%,92.9%,22.8%	(32,7,968) -128.6%,75.0%,13.7%		
	Sol3	(49,9,524) -63.3%,65.4%,30.9%	(16,30,707) 46.7%,-15.4%,6.7%		Sol3	(51,7,824) -264.3%,75.0%,26.6%	(34,8,956) -142.9%,71.4%,14.8%		
	Sol4	(43,9,529) -43.3%,65.4%,30.2%	(21,27,741) 30.0%,-3.8%,37.9%		Sol4	(58,10,809) -314.3%,64.3%,27.9%	(31,8,973) -121.4%,71.4%,13.3%		
	Sol5	(30,6,591) 0.0%,76.9%,22.0%	(23,27,713) 23.3%,-3.8%,5.9%		Sol5	(64,10,795) -357.1%,64.3%,29.1%	(32,8,966) -128.6%,71.4%,13.9%		
	Sol6	(34,6,583) -13.3%,76.9%,23.1%	(17,29,707) 43.3%,-11.5%,6.7		Sol6	(61,9,799) -335.7%,67.9%,28.8%	(27,9,1060) -92.9%,67.9%,5.5%		
	Sol7	(32,6,584) -6.7%,76.9%,23.0%	(17,28,711) 43.3%,-7.7%,6.2%		Sol7	(61,9,797) -335.7%,67.9%,29.0%	(31,9,969) -121.4%,67.9%,13.6%		
	Sol8	(49,9,525) -63.3%,65.4%,30.7%	-	MM	Sol8	(56,7,815) -300.0%,75.0%,27.4%	(31,7,977) -121.4%,75.0%,12.9%		
	Sol9	(51,9,517) -70.0%,65.4%,31.8%	-		Sol9	(56,7,811) -300.0%,75.0%,27.7%	(29,7,1017) -107.1%,75.0%,9.4%		
	Sol10	(34,6,570) -13.3%,76.9%,24.8%	-			-	-	-	
	Sol11	(38,6,560) -26.7%,76.9%,26.1%	-			-	-	-	
	Sol12	(36,6,561) -20.0%,76.9%,26.0%	-			-	-	-	
	Sol13	(55,9,516) -83.3%,65.4%,31.9%	-			-	-	-	



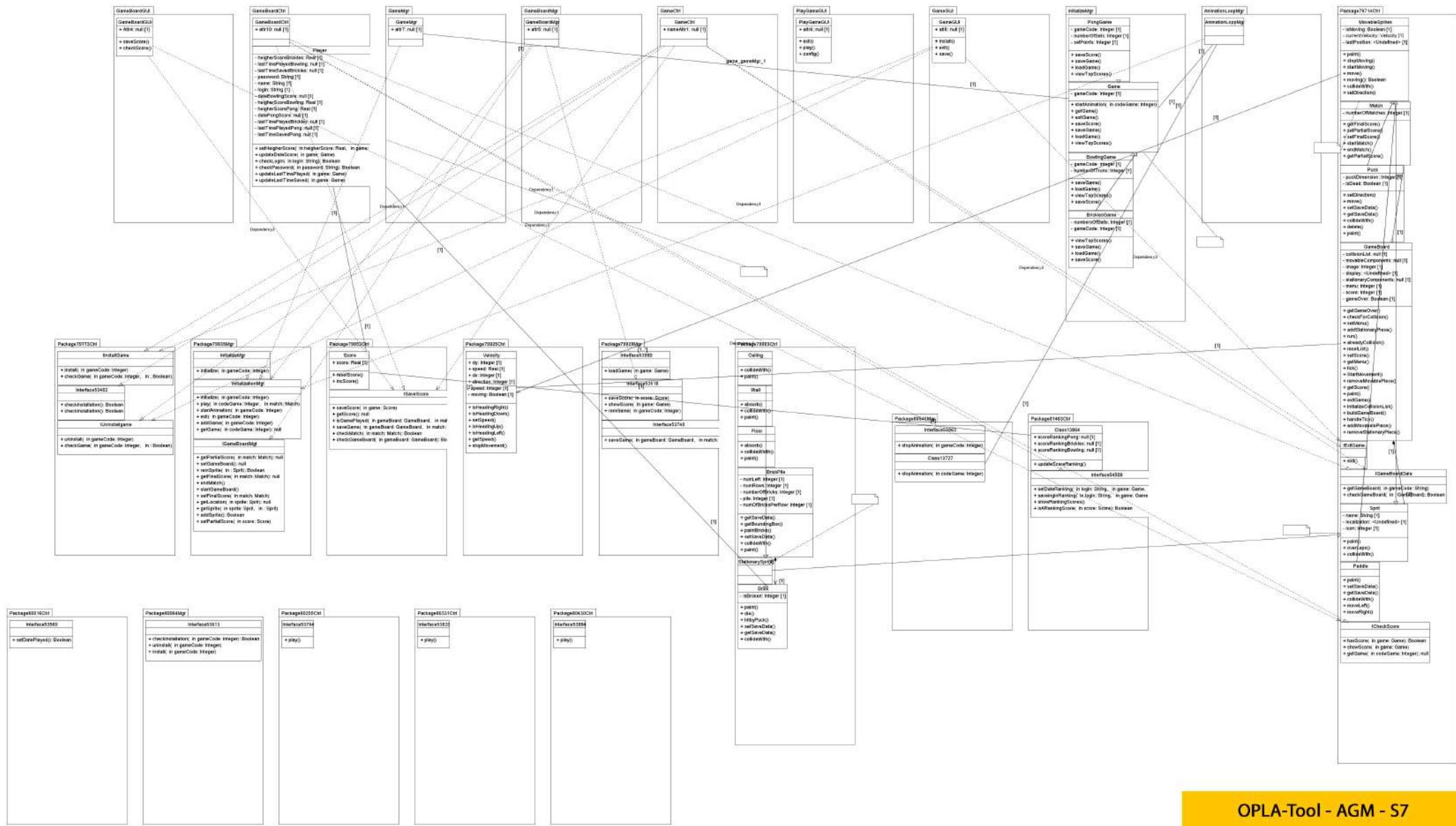






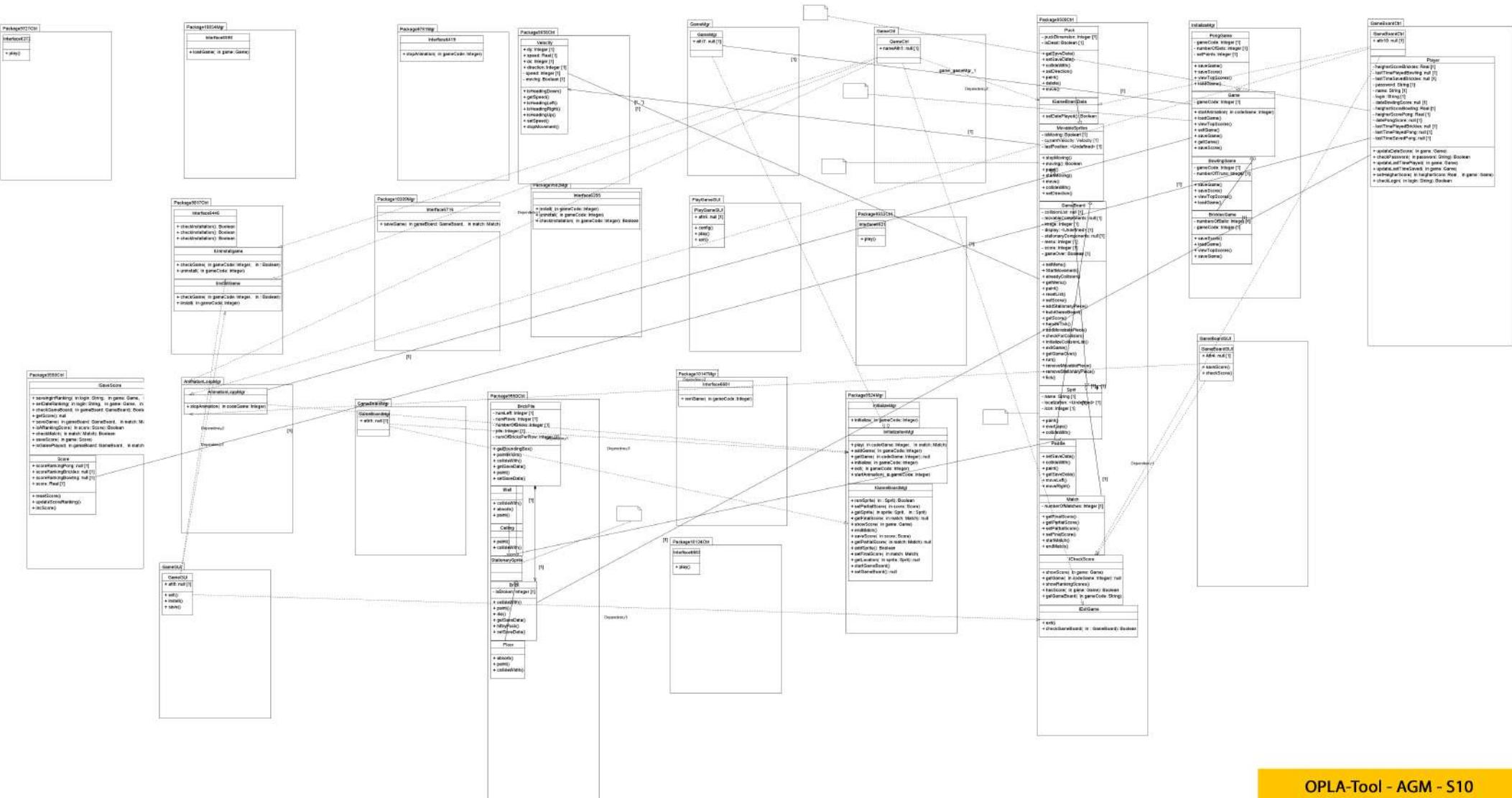


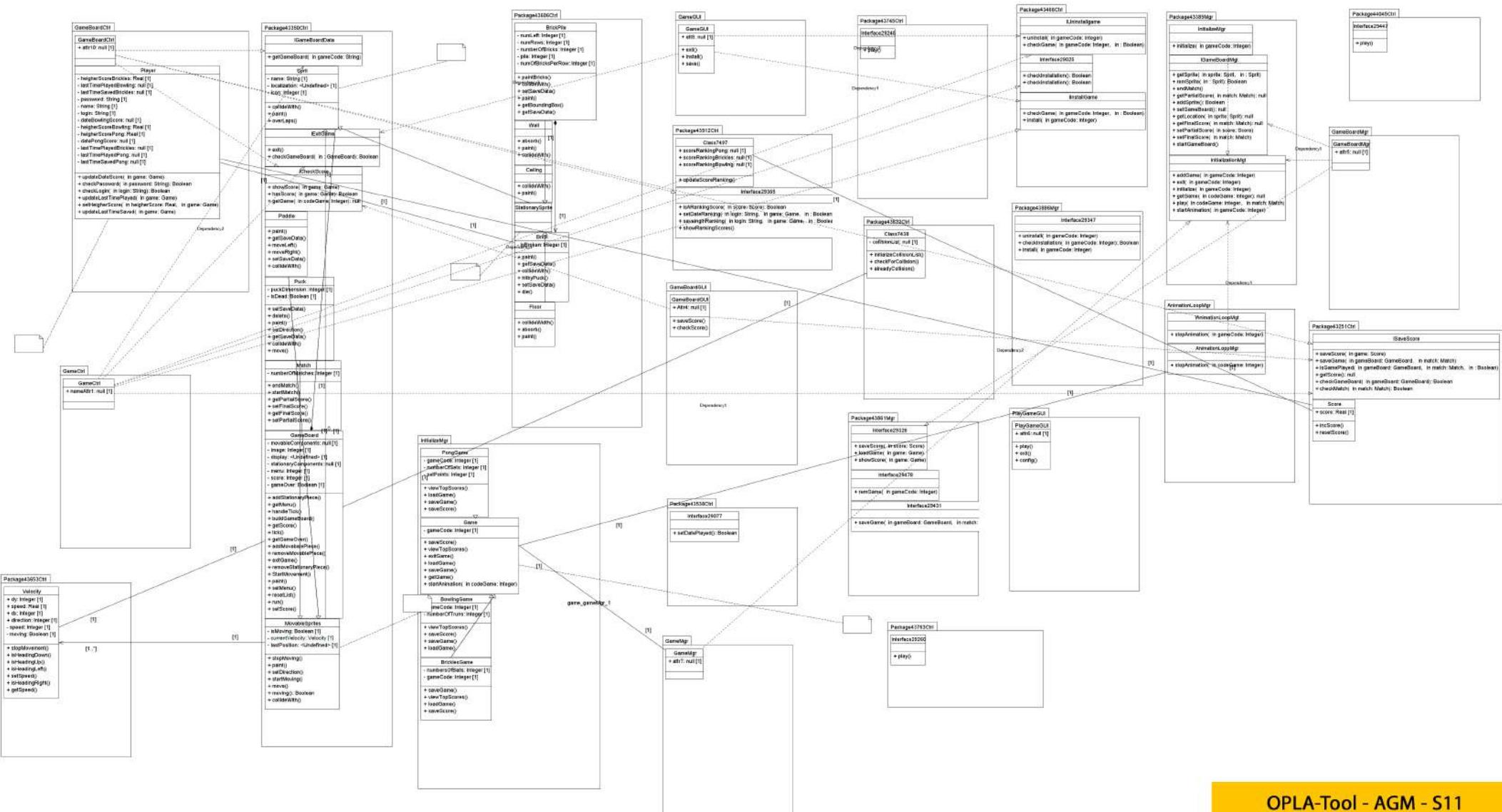






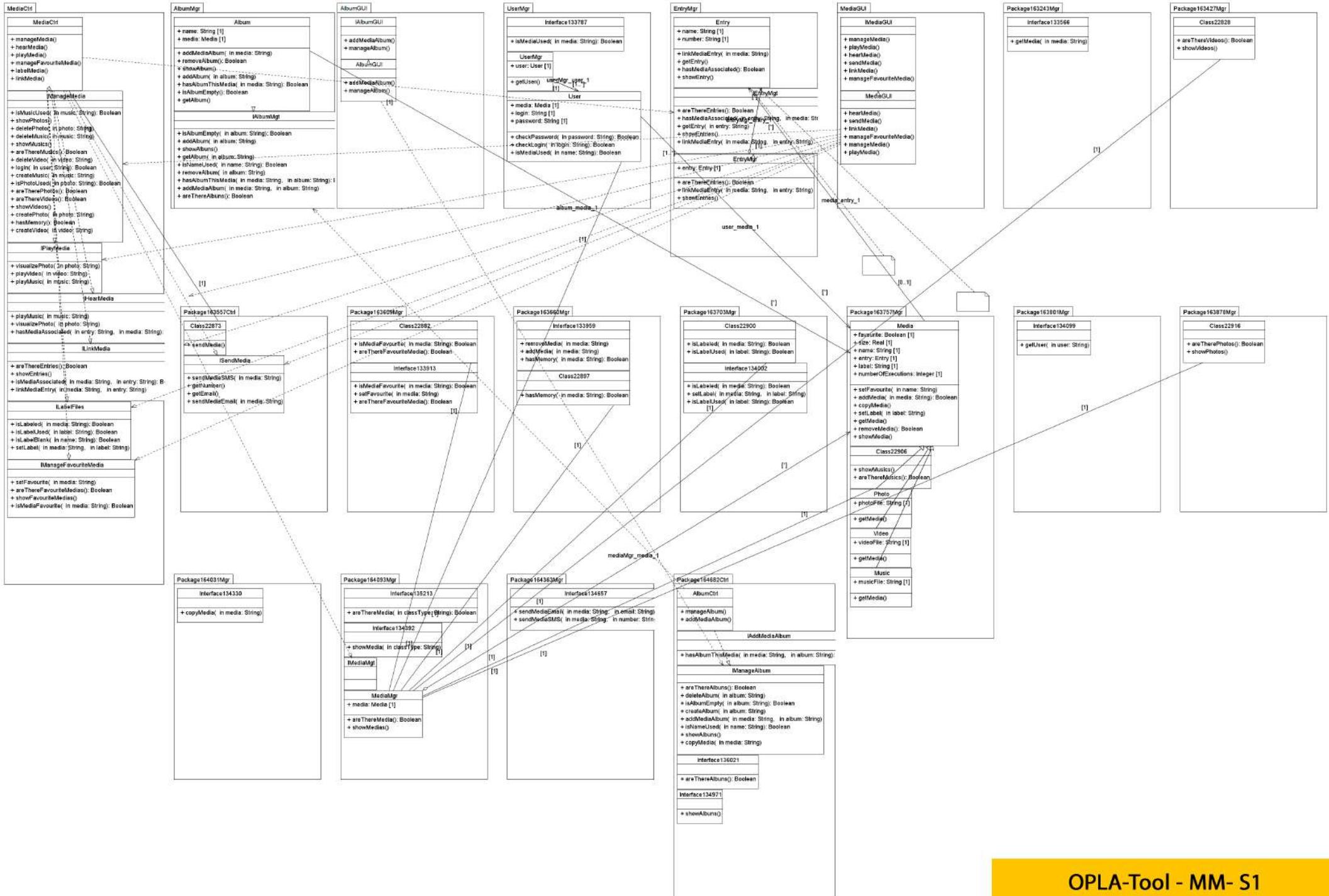


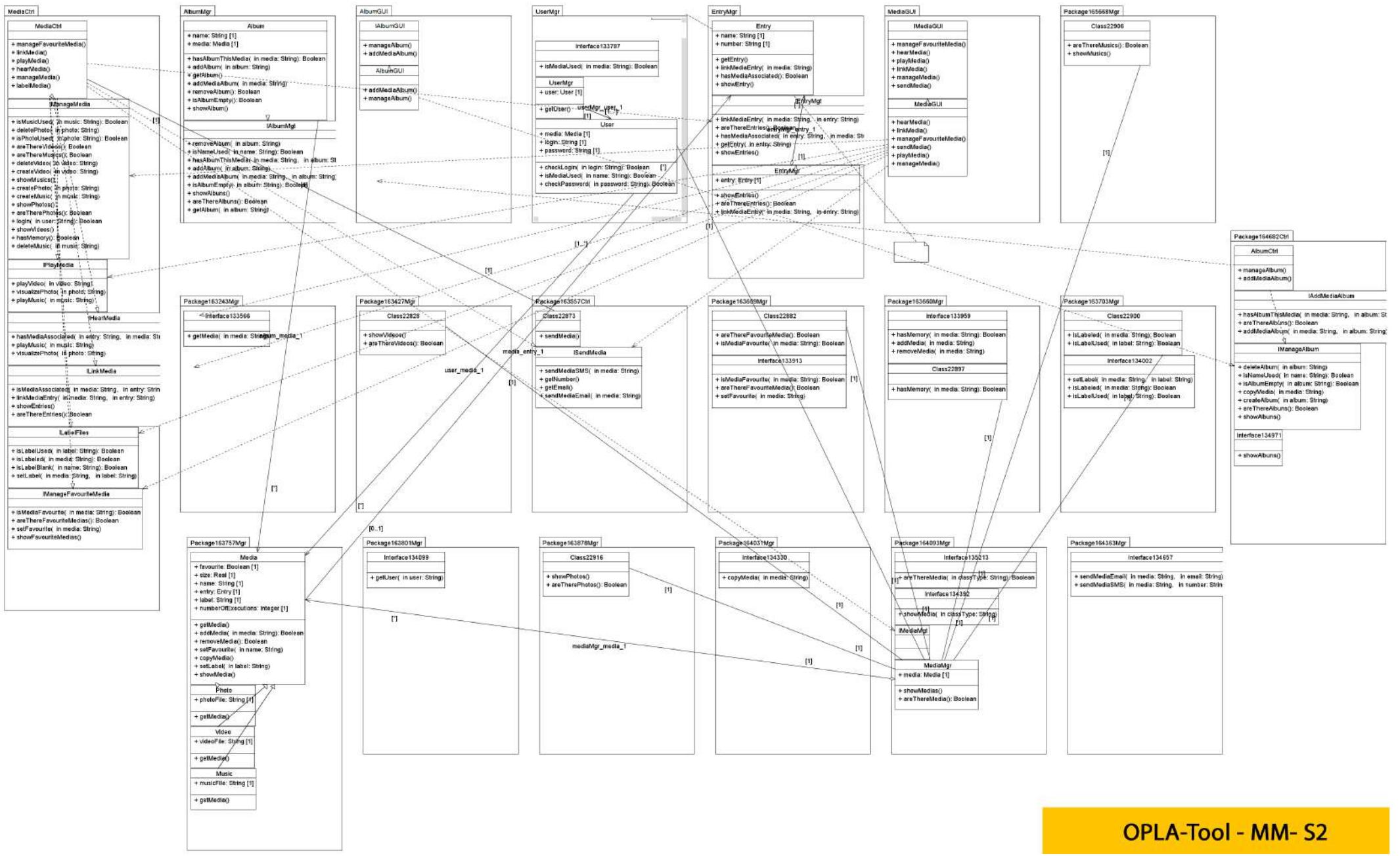


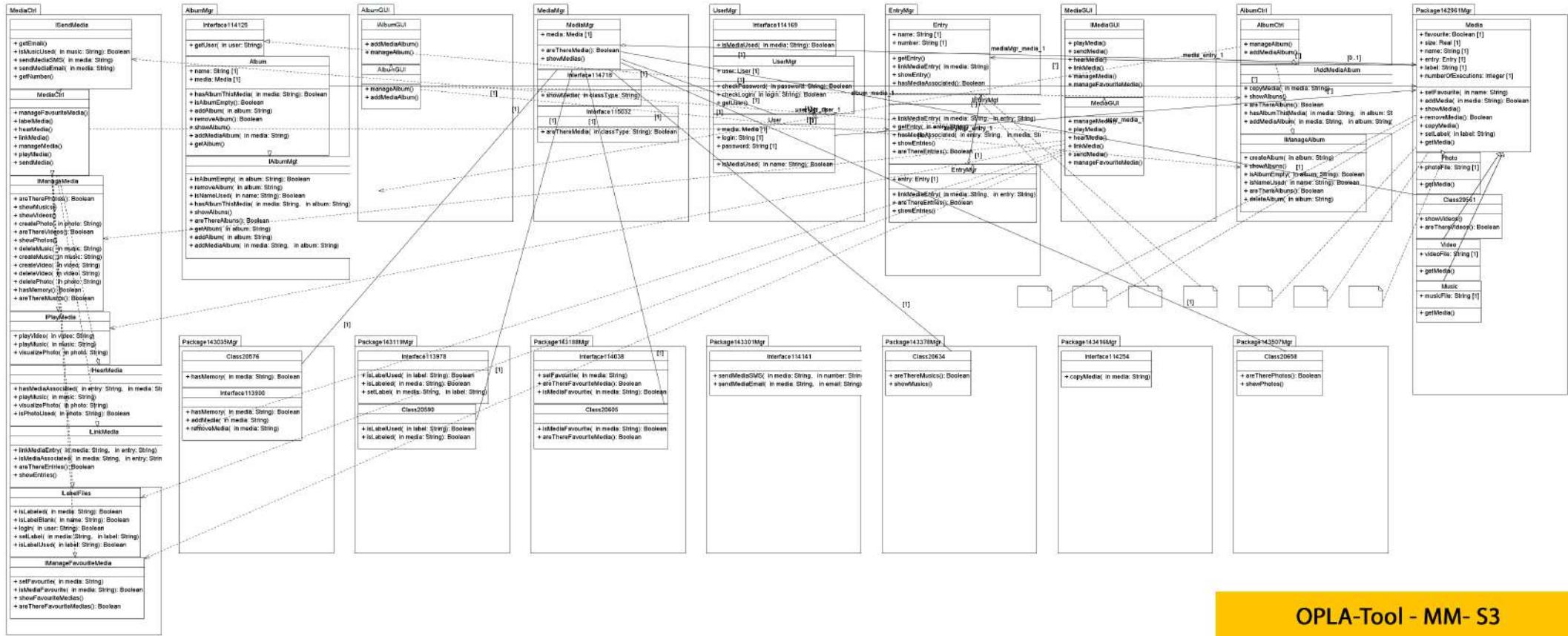




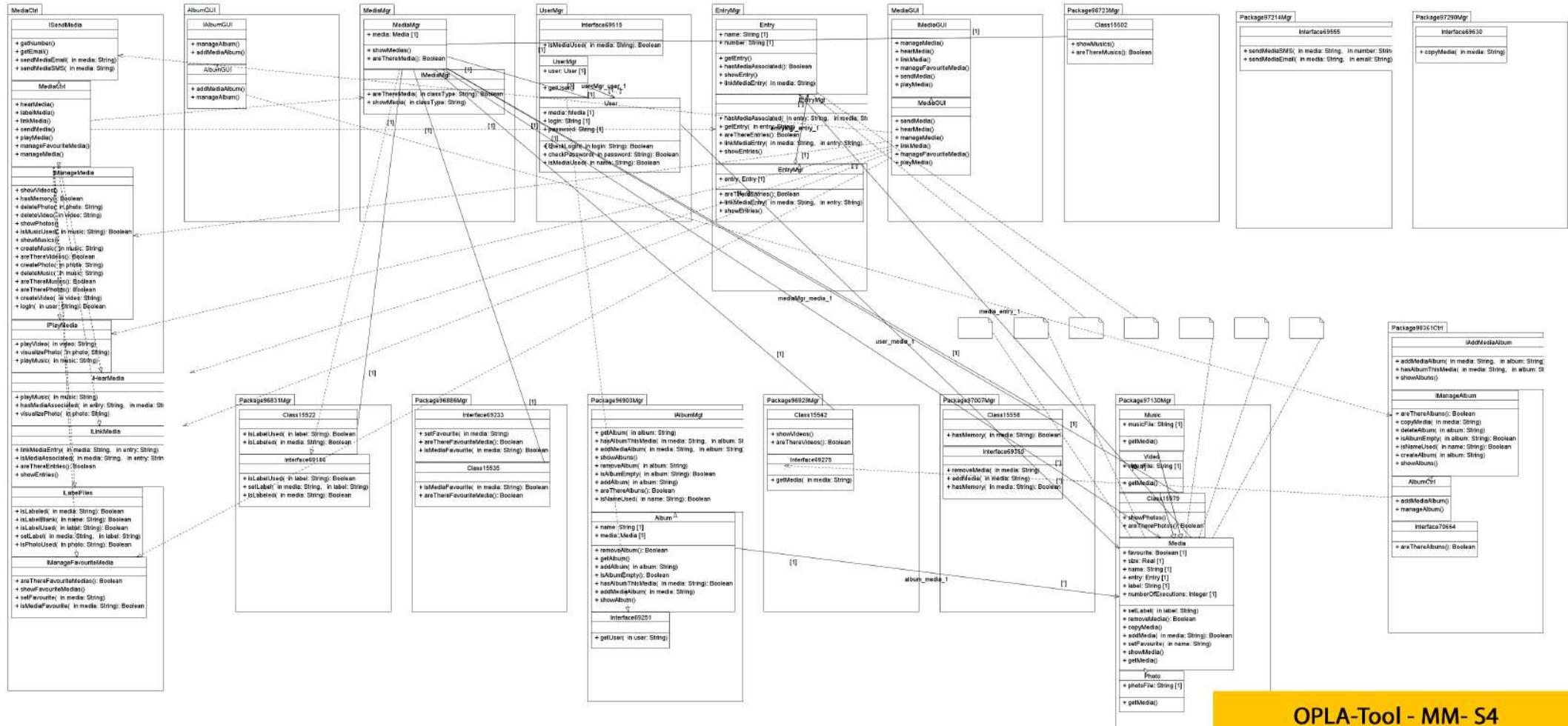


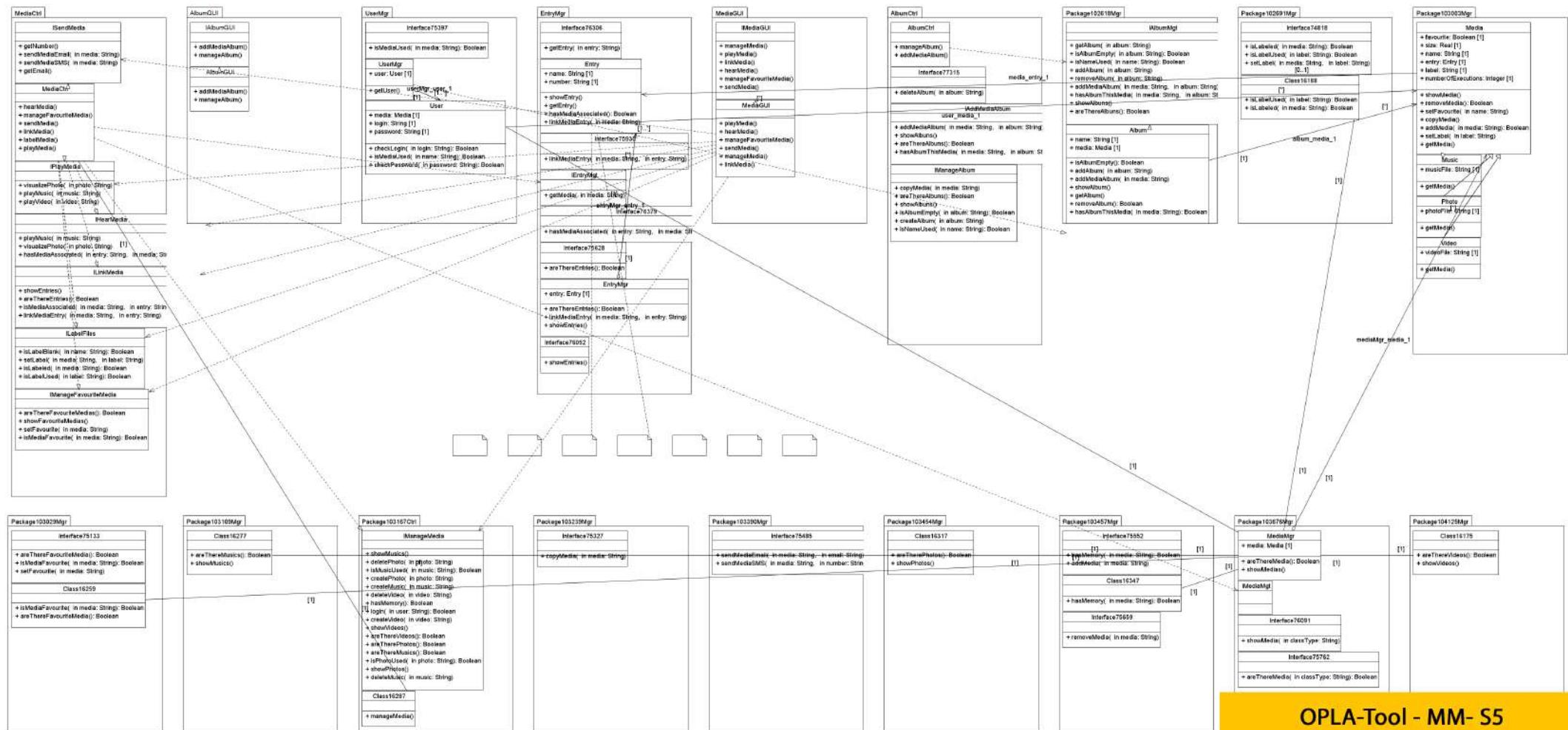




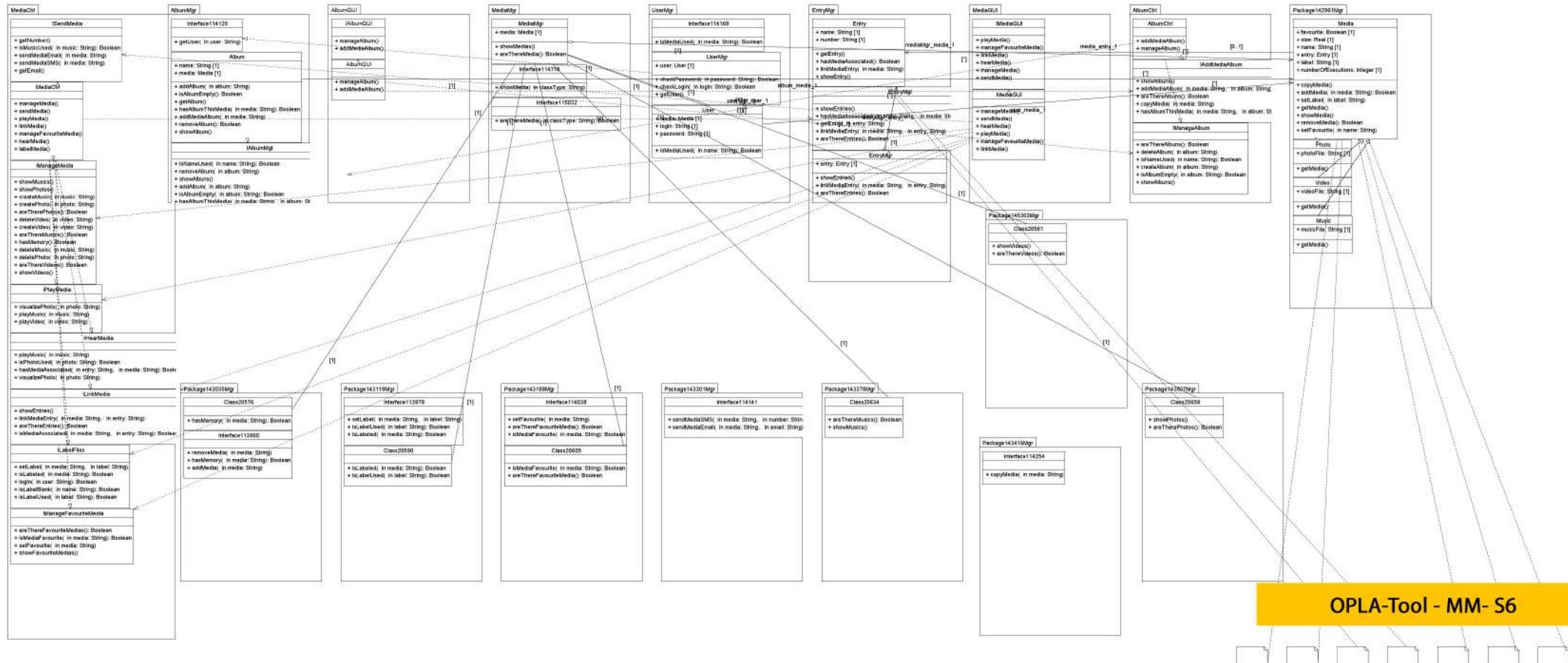


OPLA-Tool - MM- S3

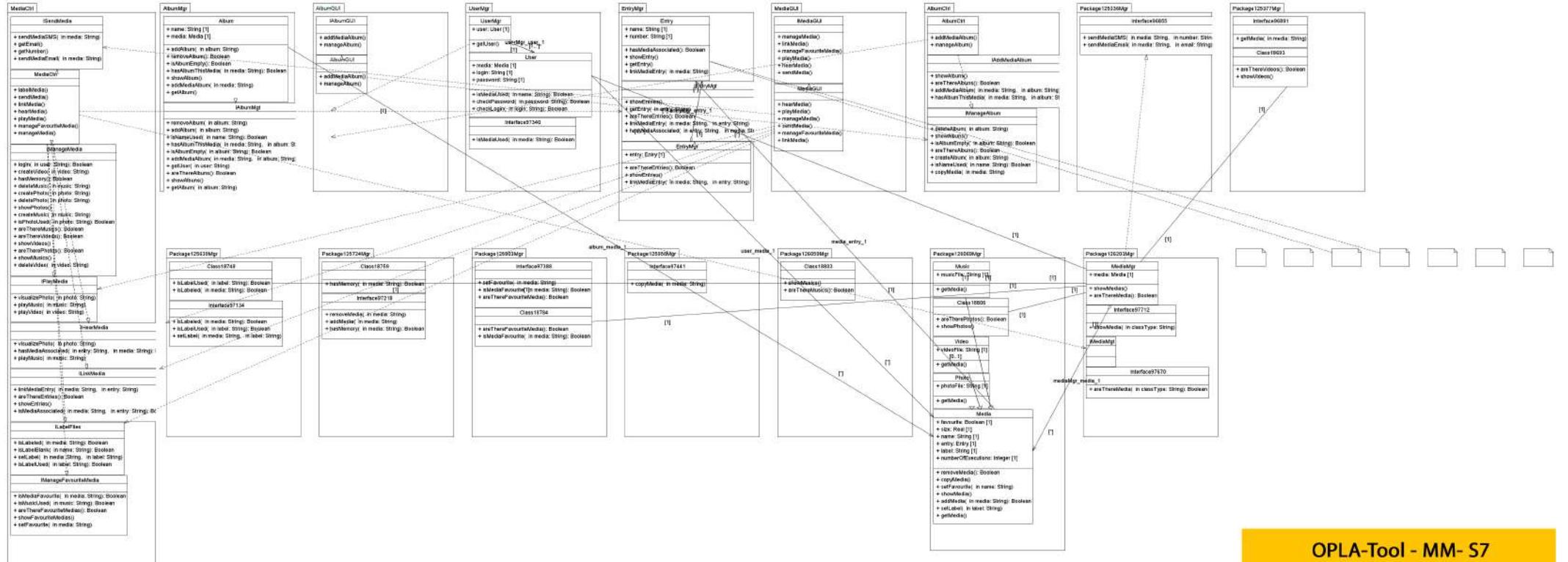




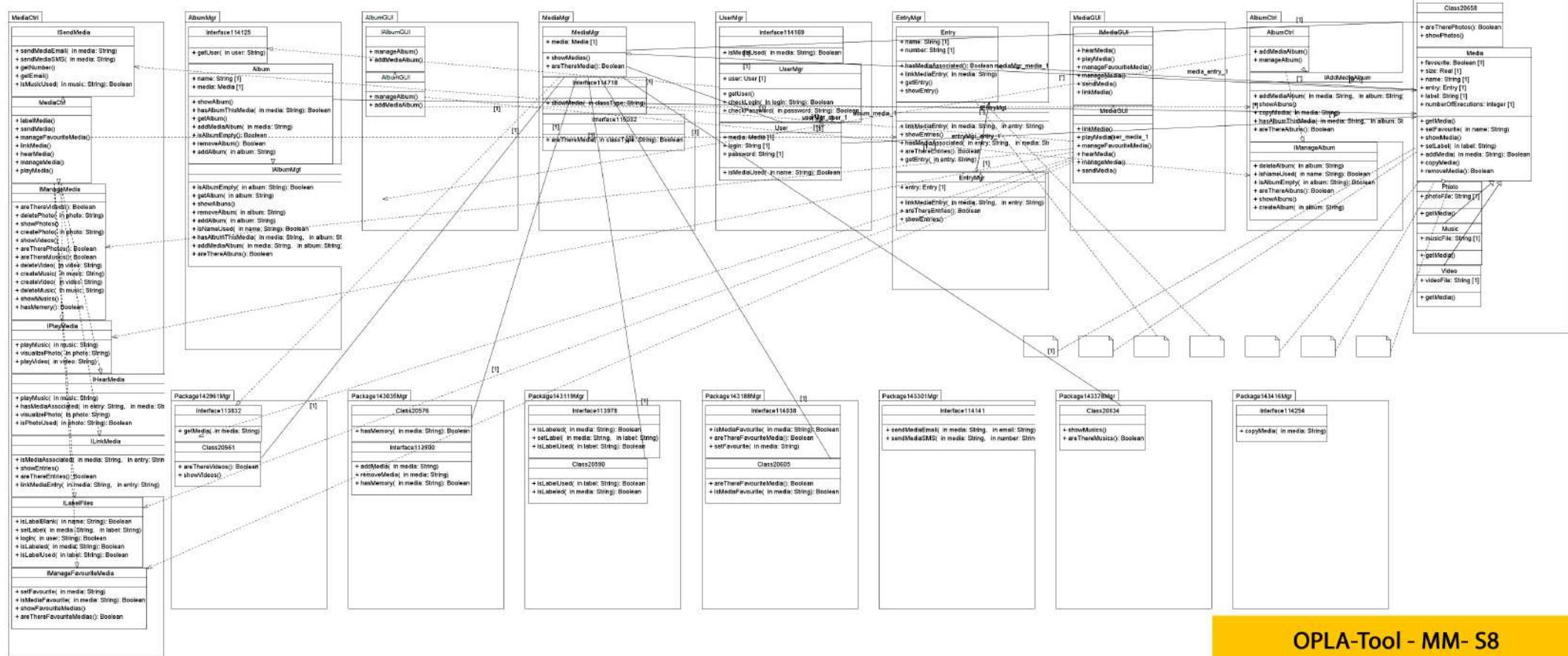
OPLA-Tool - MM- S5



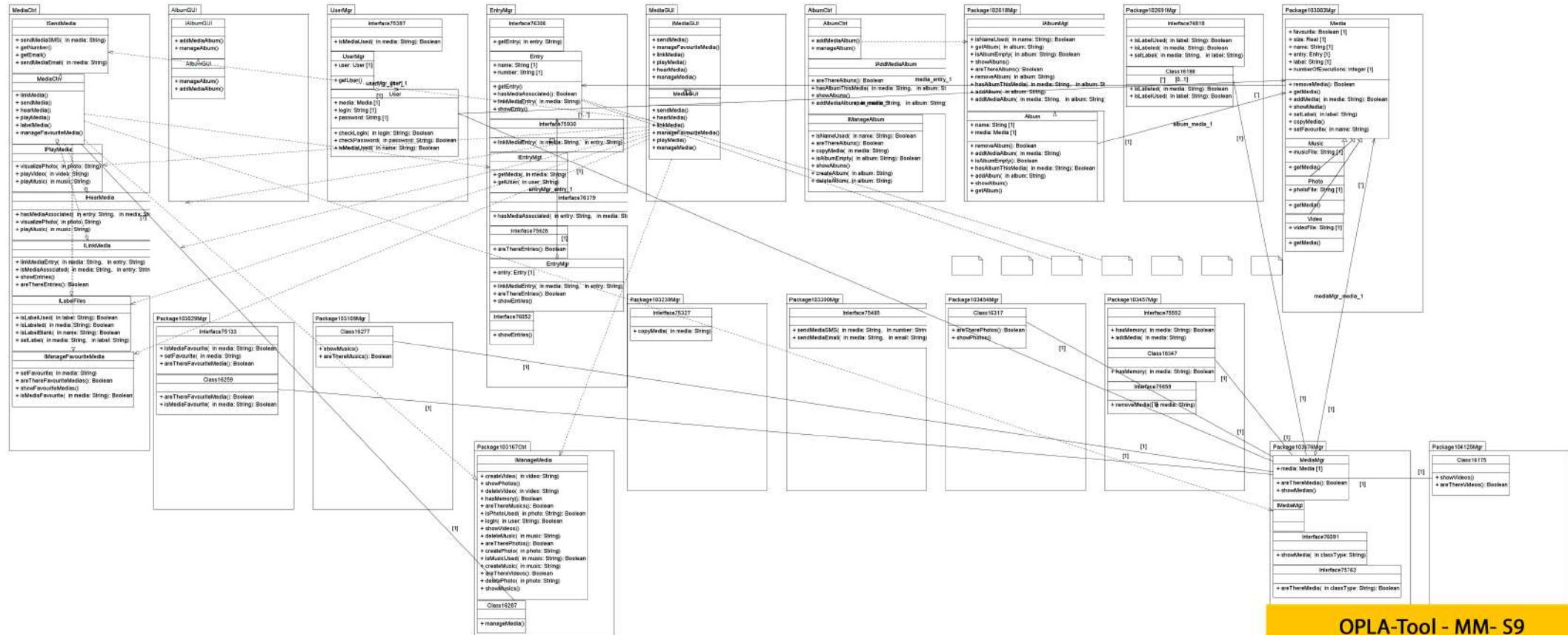
OPLA-Tool - MM- S6

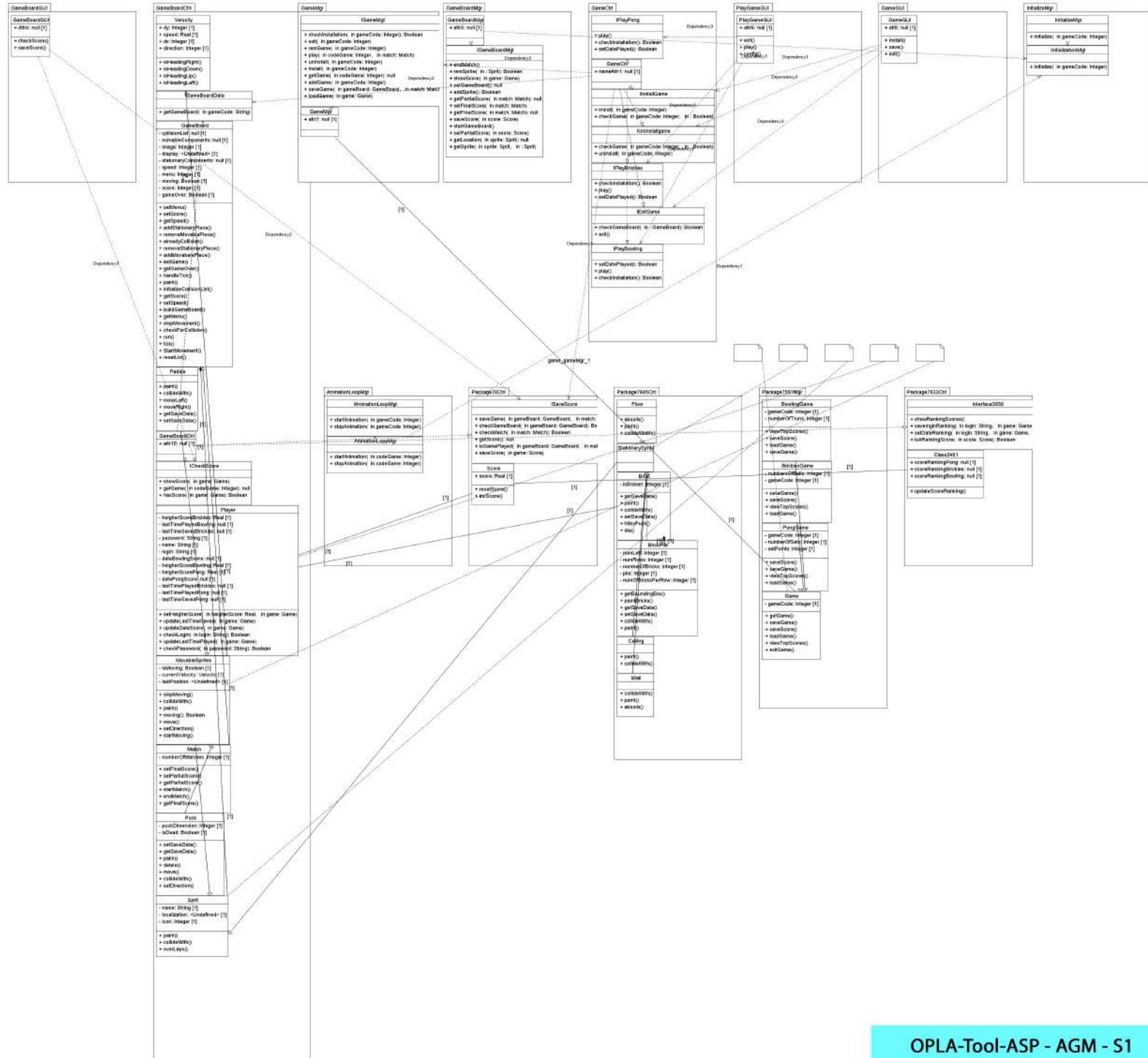


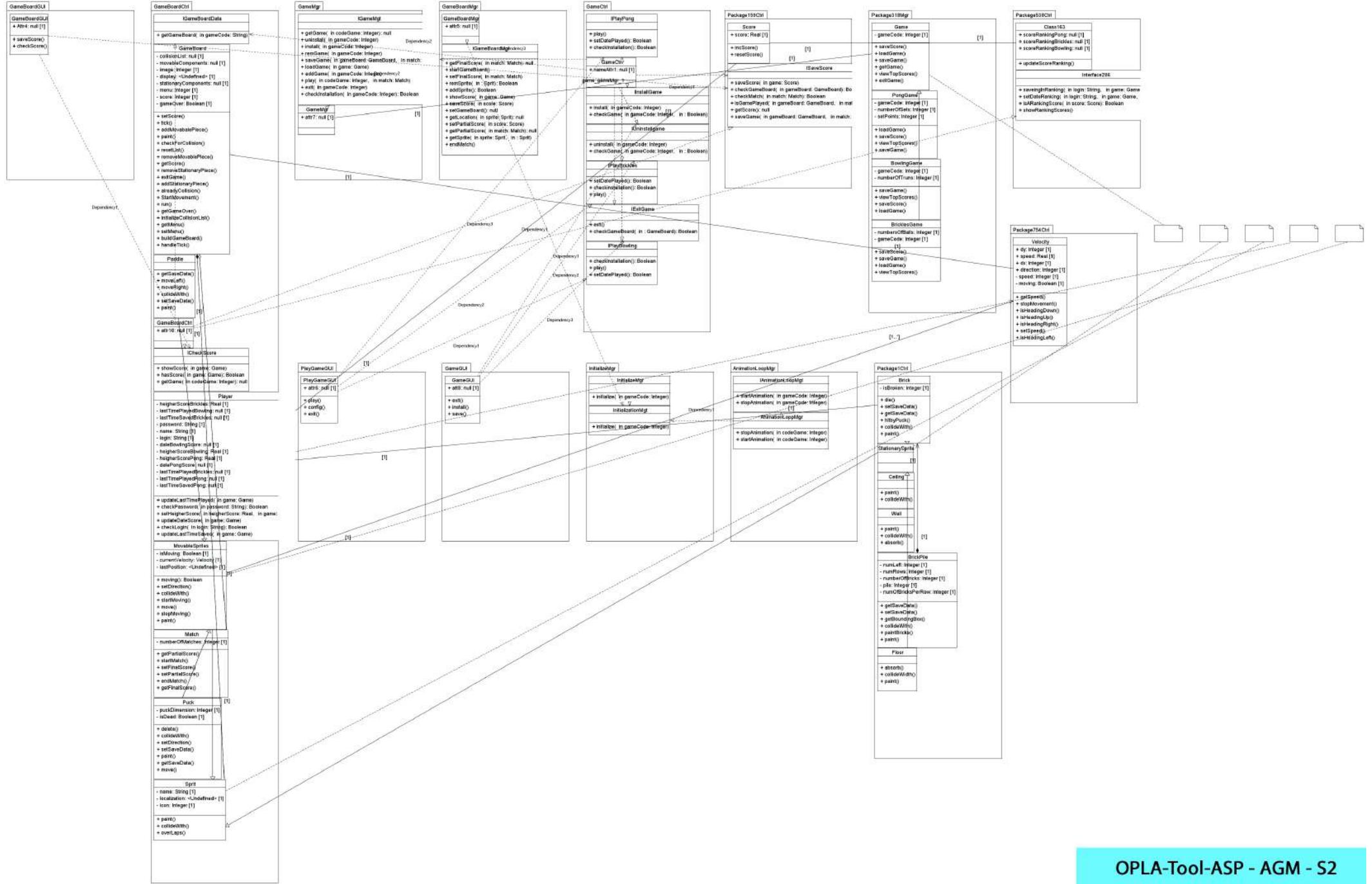
OPLA-Tool - MM- S7

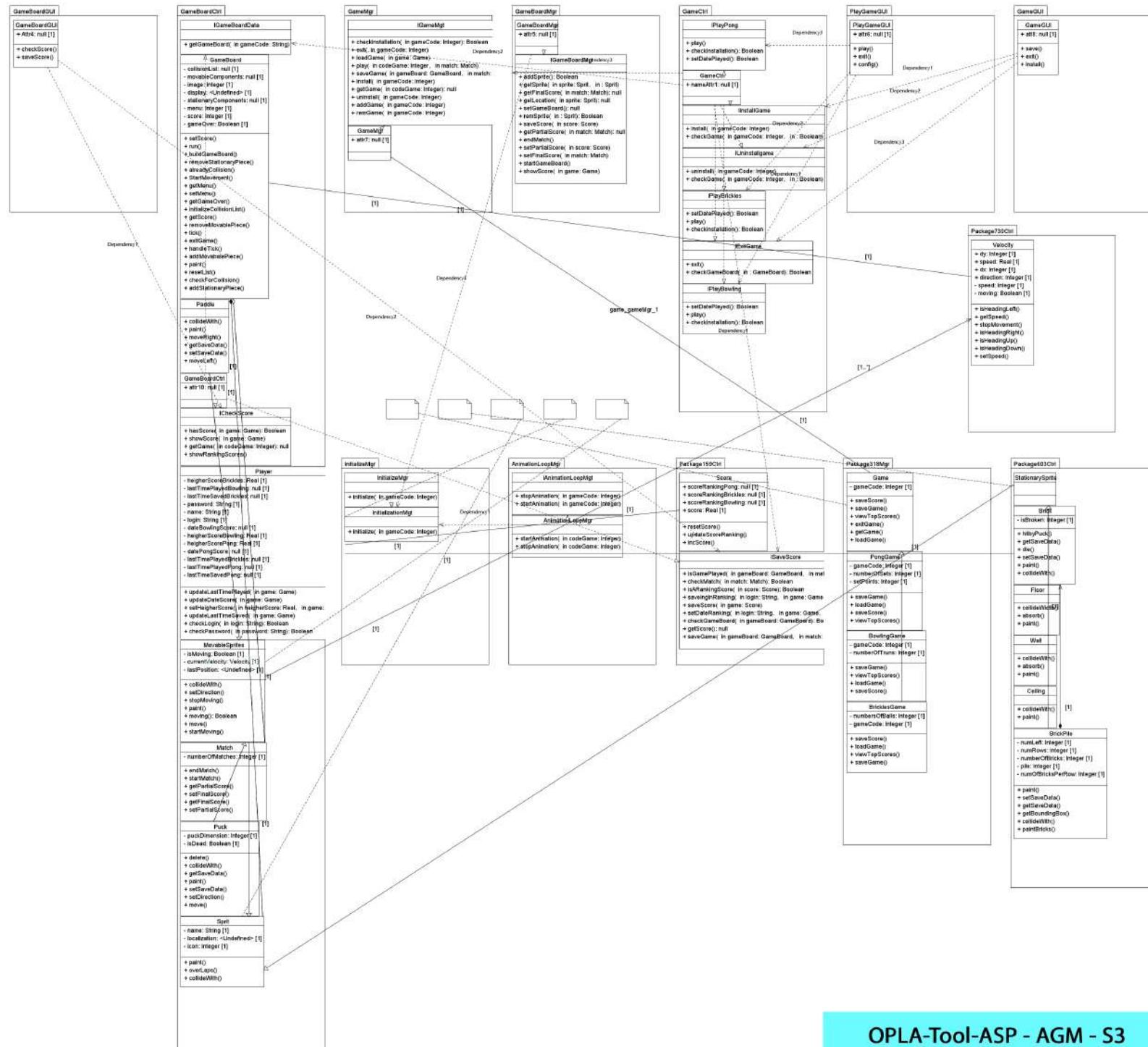


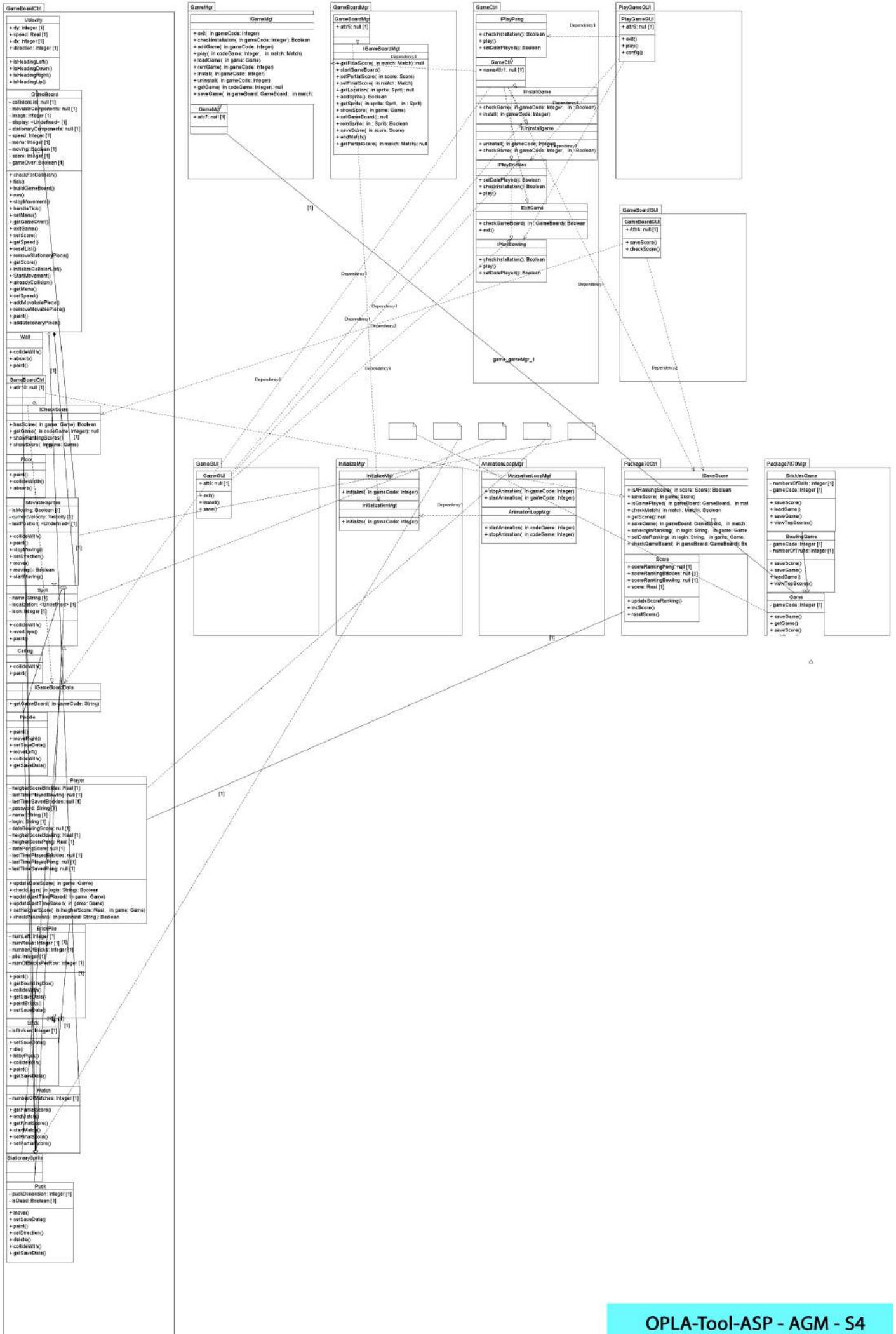
OPLA-Tool - MM- S8

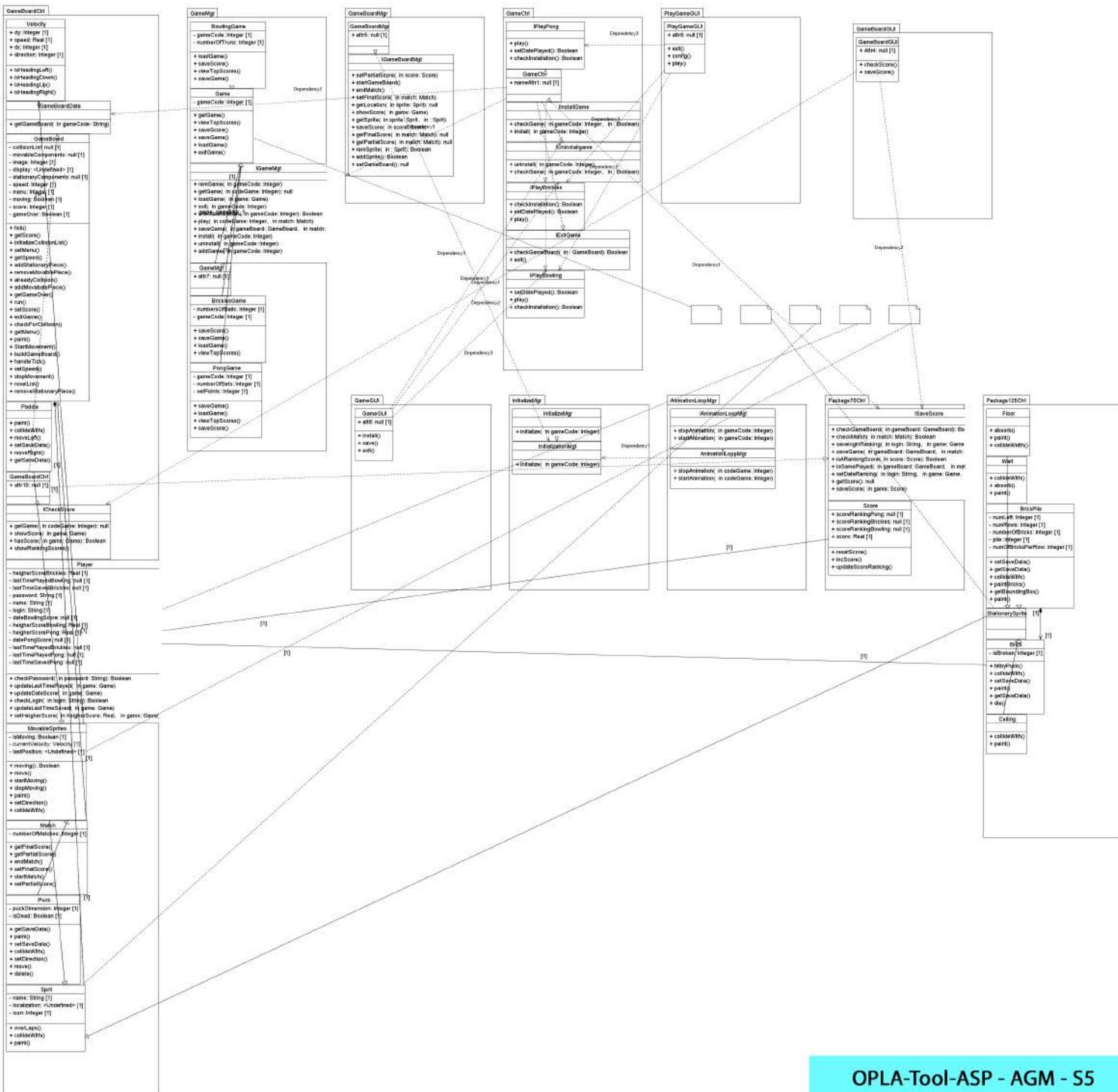


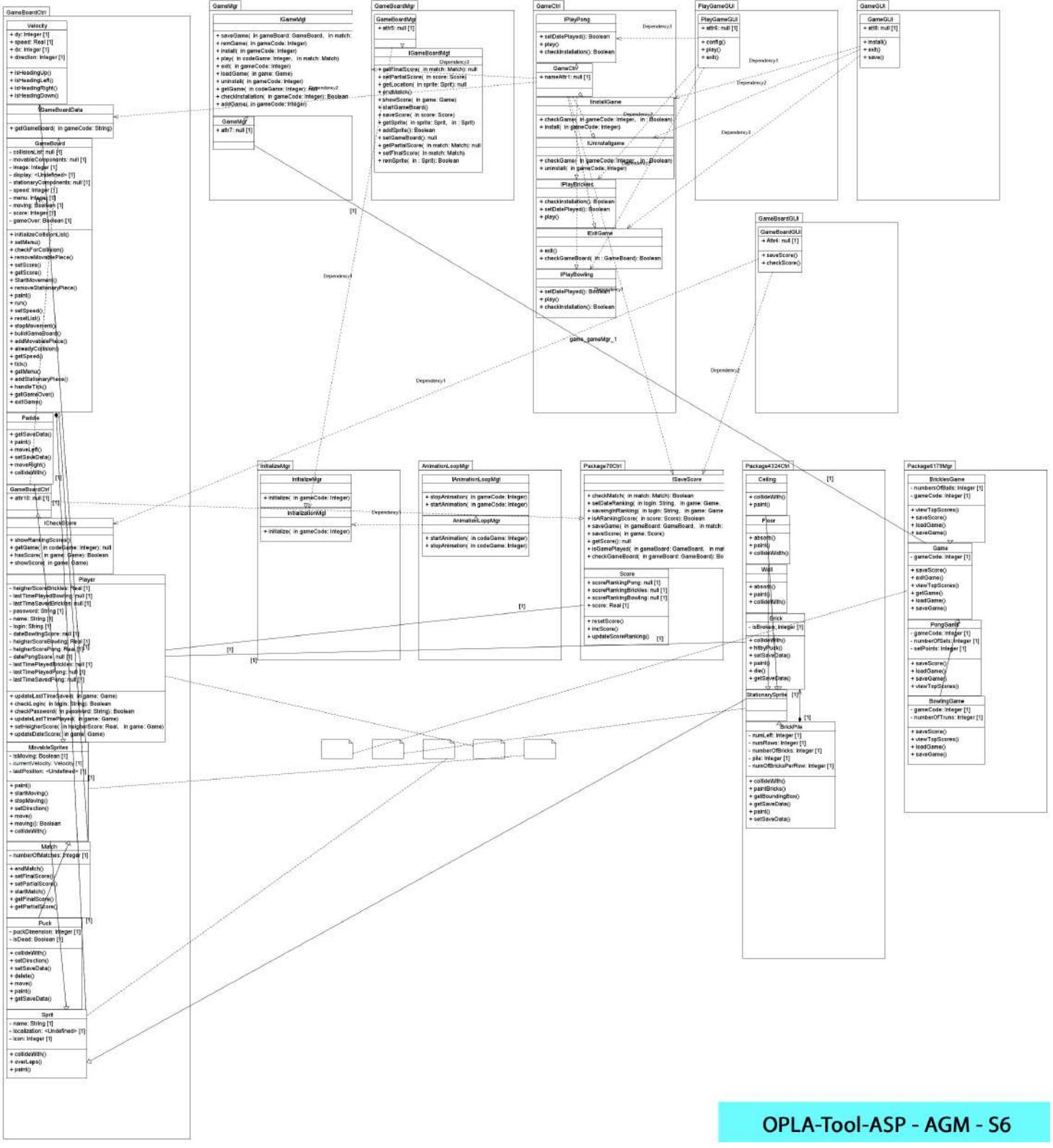


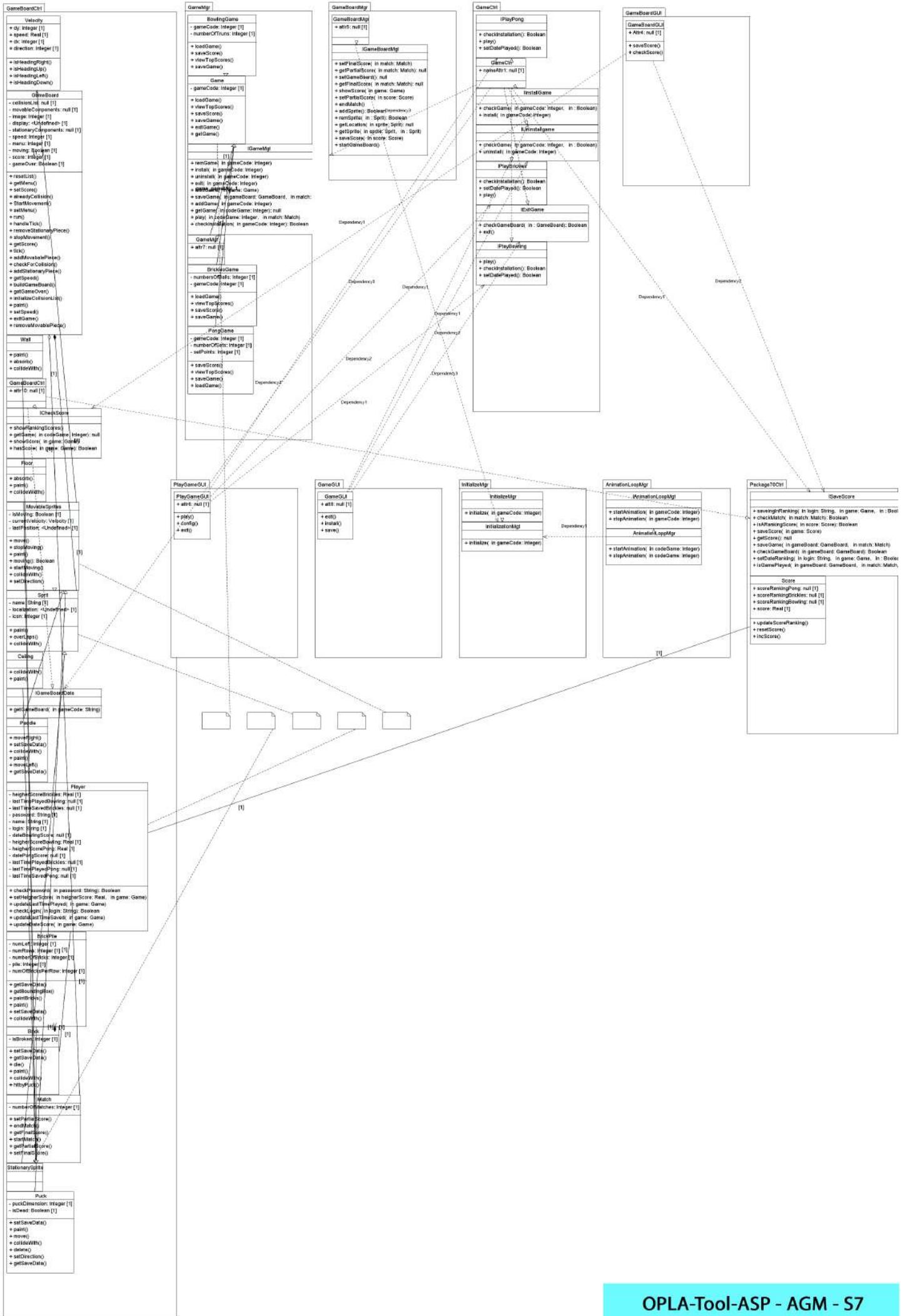


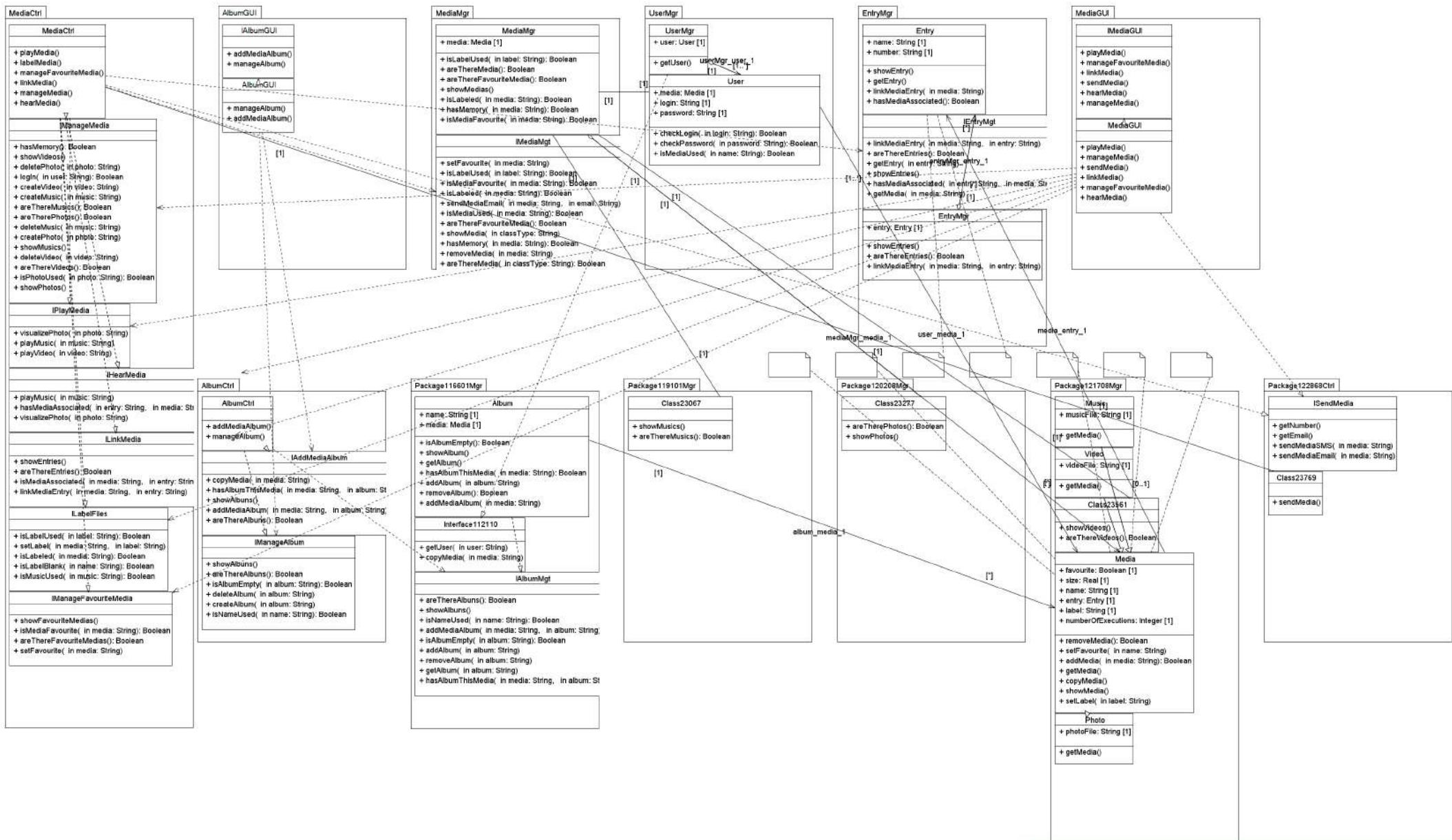


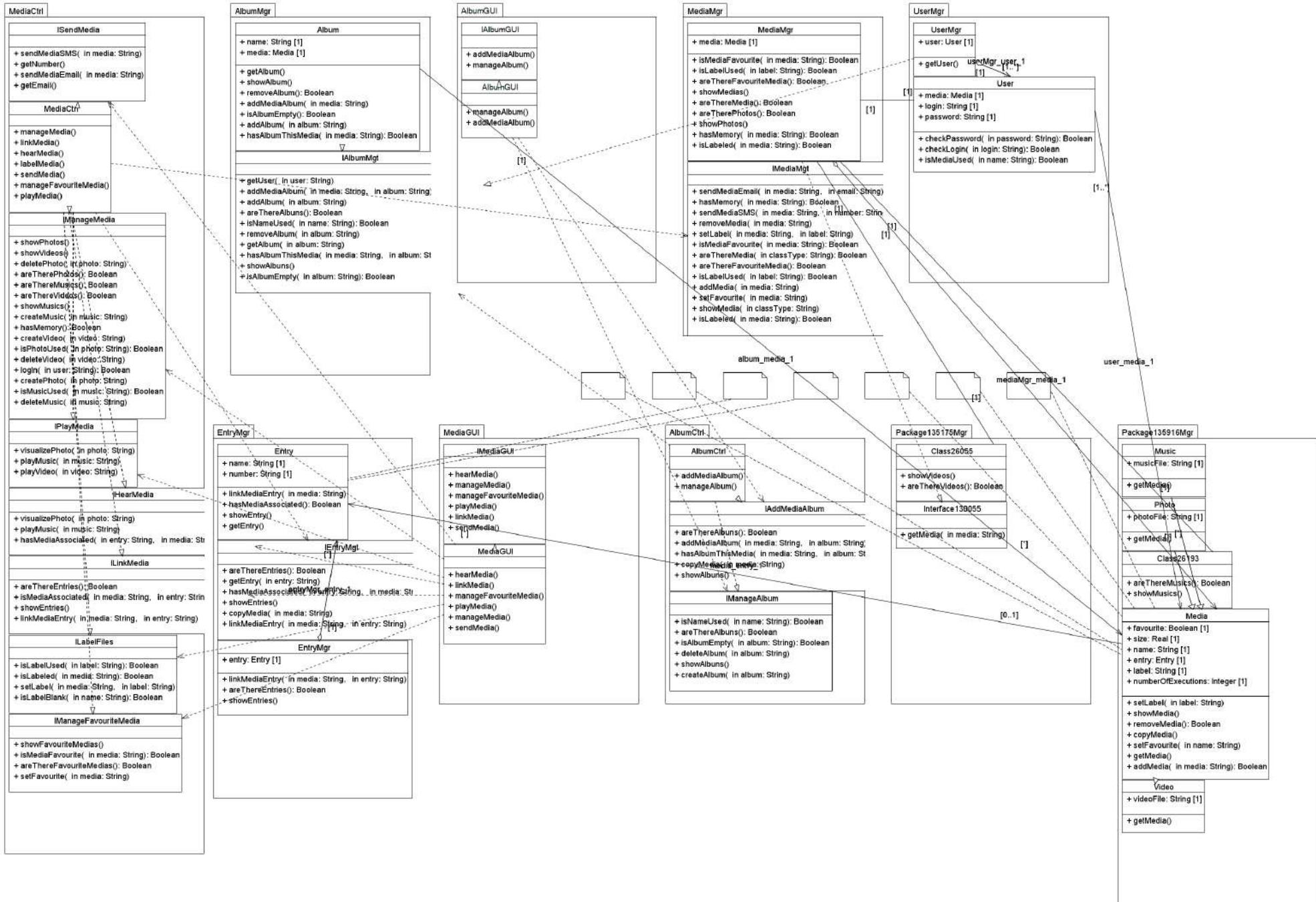


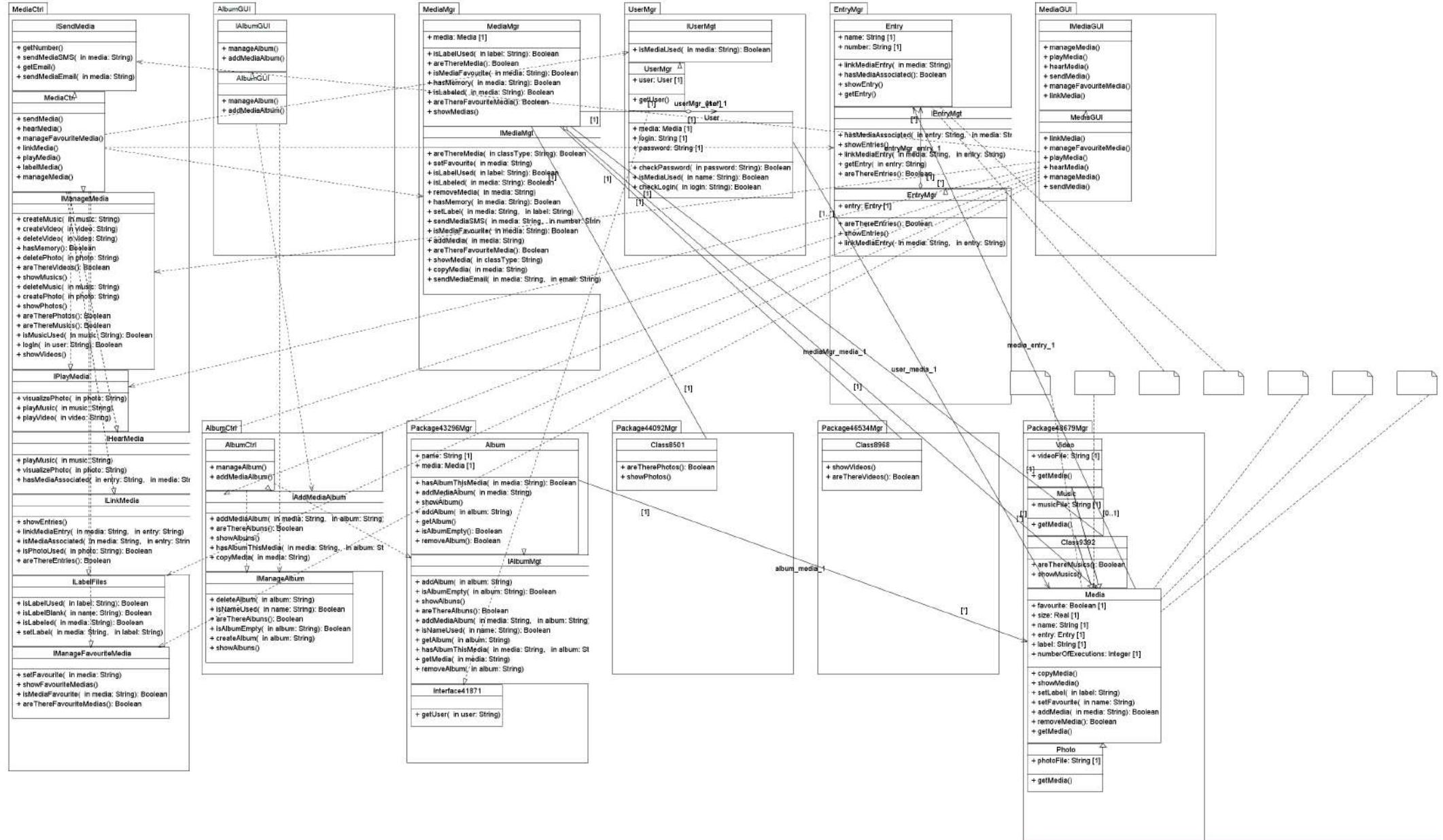


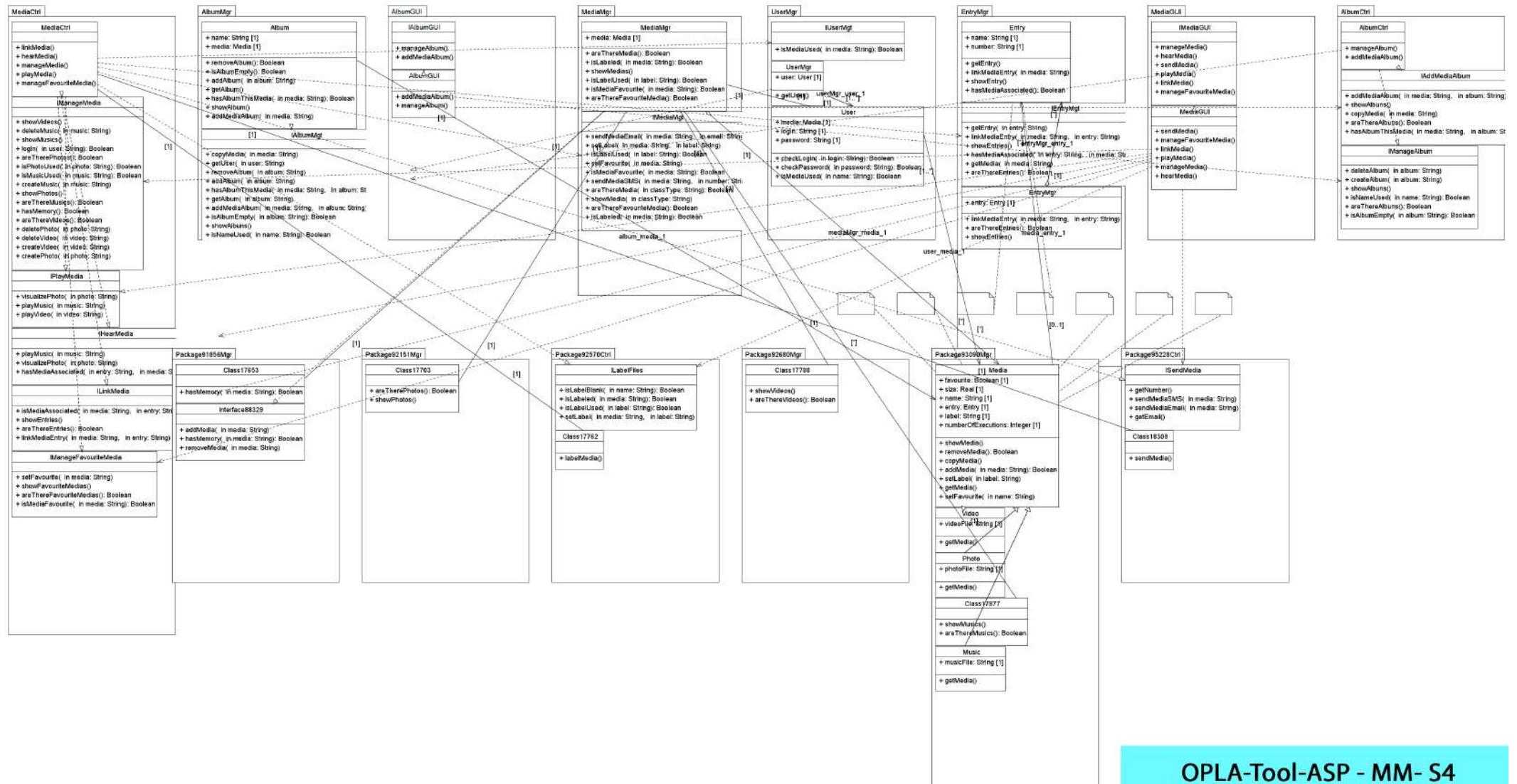












OPLA-Tool-ASP - MM- S4

