

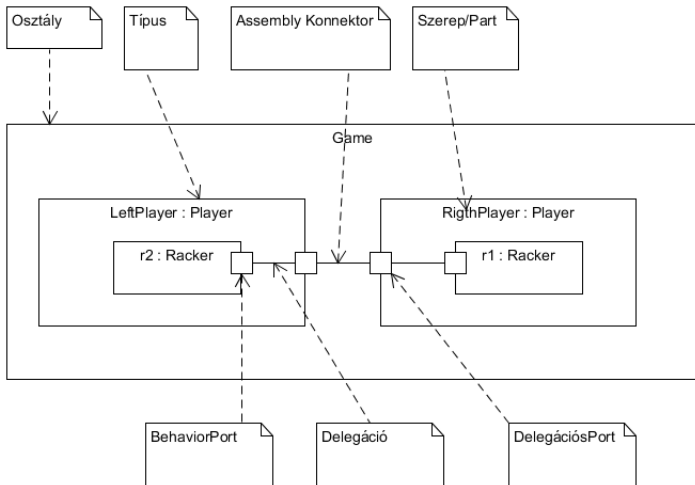
# Komponens-alapú UML modellek fordításának vizsgálata

Nagy András

2019. január

- *txtUML* keretrendszerben írjuk le a komponens-alapú modellt, Java-szerű nyelven, mely végrehajtható.
- Lefordítható egy szabványos UML2 modellre.
- A cél a kompozit struktúrák és akciók megfelelő UML2-es szabványának megtalálása, melyből hatékony C++ kódot szeretnénk generálni.

# Példa egy konkrét kompozit struktúrára



- Port típusának reprezentálása, elvárt és szolgáltatott interfészek kifejezése.
- Két port összekapcsolása futási időben.
- Porta való üzenetküldés.
- Mi a szabványos modell végrehajtási szemantikája?
  - UML2 szabvány.
  - *FUML* egy részhalmazához ad precíz szemantikát.
  - *PSCS* szabvány a kompozitokra próbálja kiegészíteni.

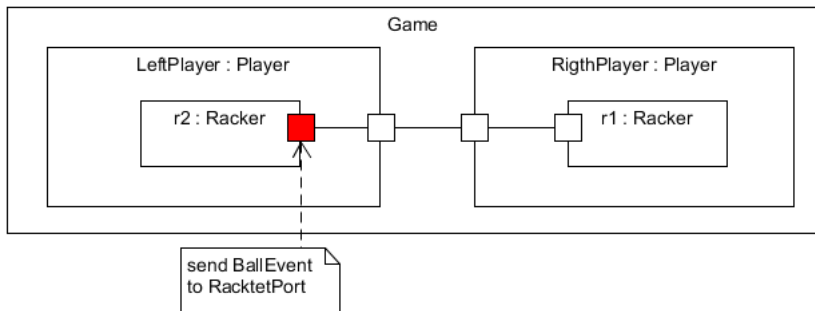
## UML specifikáció szerint

*A Connector specifies links (see 11.5 Associations) between two or more instances playing owned or inherited roles within a StructuredClassifier." "A CreateLinkAction is a LinkAction for creating links."*

- A *Connector* értelemszerű (de mi az a *type* referencia?)
- A problémás a *connect* művelet
  - Nincs *connect* akció UML-ben
  - *DefaultConstructionStrategy* (de az a szerkezet nem mindig egyértelmű..)
  - A *CreateLinkAction* segítségével összeköthetünk két portot a konnektor típusa mentén (Itt jön be a *type* referencia, mely egy asszociációnak felel, ezt még ki kell generálni.).

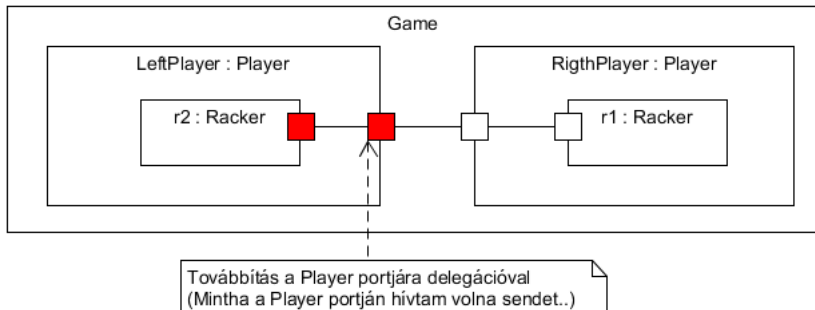
- Sok alternatíva (típusbiztonság, adatrepresentációs különbségek, stb.)
- Különböző elvárások a generált kóddal szemben (hatékonyság, olvashatóság, külső kóddal történő biztonságos illesztés, stb.)
- Ez a futási idejű könyvtár kialakításából és kódgenerálásból áll az UML szemantika alapján.
- Ezeket a szempontokat figyelembe véve a munka elemzi az egyes kompozit elemek kódgenerálási lehetőségeit.

# Üzenetáram eleje



Kérdések: Hogyan alakítsuk ki a port osztályt C++-ban, hogyan küldünk rá üzenetet?

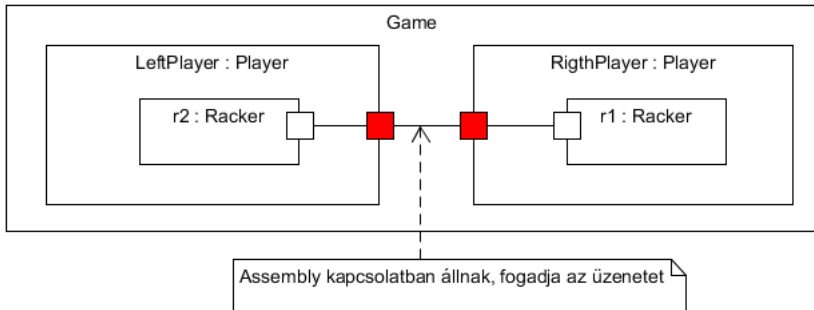
# Üzenet továbbítása a szülő komponens felé - Delegáció



Problémák: Üzenet továbbítása szülő felé delegációs kapcsolat mentén.

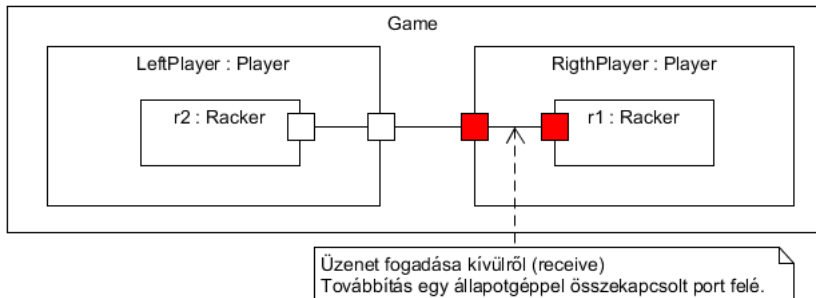


# Üzenet átadása testvér komponensek - Assembly



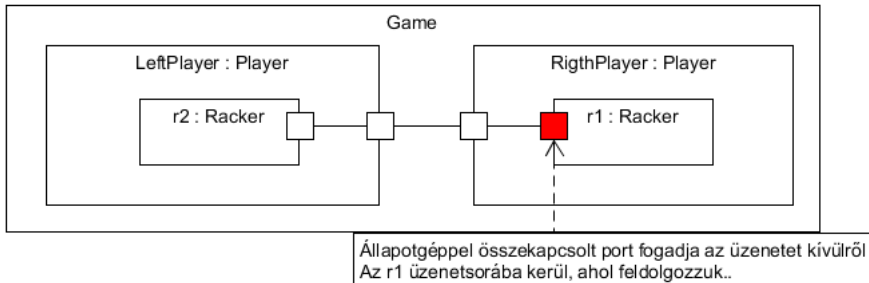
Problémák: Más a *send* viselkedése, mint delegáció esetén, a port elvárt interfészének kell megfelelni.

# Üzenet továbbítása a gyerek komponensek



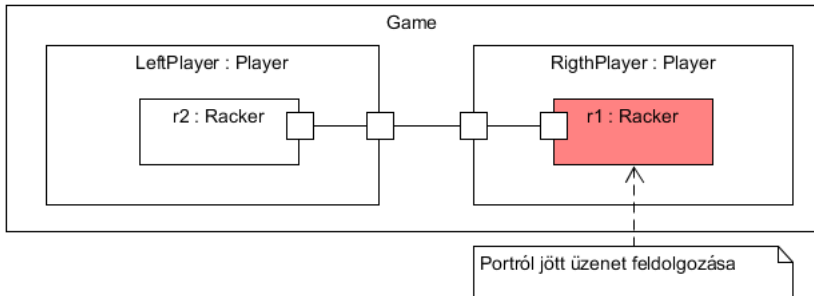
Delegációs kapcsolat másik irányba, a gyerek komponense felé továbbítom az üzenetet.

# Gyerek felé továbbítás delegáció esetén



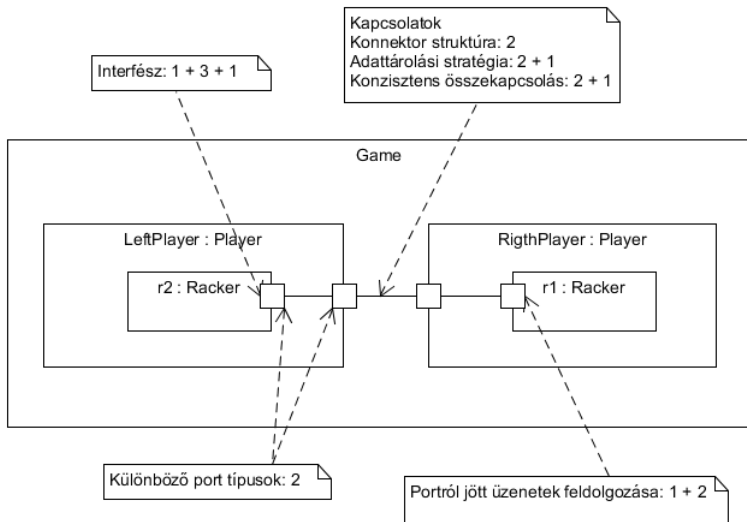
A gyerek komponens fogadó portja másképp kell, hogy viselkedjen, mint a szülőé, mivel állapotgéppel összekapcsolt port.

# Portról jött üzenet feldolgozás



Az üzenet az üzenetsorba kerül, fel kell dolgoznunk. Honnan tudjuk, melyik portról jött, átmeneteket kiterjesztése, hogy portról jött üzeneteket is kezeljen.

# Elemezett kódgenerálási stratégiák száma



# Példa: Interfész kódgenerálási stratégiák

Nincs interfész, a forráskód validációja kiszűri a nem típusbiztos üzenetküldéseket.

Előnyök:

- Triviális, nem kell dolgozni érte.
- Hatékony.

Hátrányok:

- Külső kóddal való illesztés nem biztonságos.
- Az interfész általánosabb fogalom, nem csak portok esetén akarom használni.

## Kódja

```
class Port {  
    ...  
    void send(GeneralEvent e);  
    ...  
};
```

A fogadó szignálokat csoportosítsuk az interfészből történő leszármazással, a portnak legyen *send* művelete, ami interfész leszármazottakat vár. Hátrányok:

- Nem elég általános megoldás.
- Interfészt nem csak portok, hanem bármilyen szereplők használhatnak, de ezzel csak portokra korlátoznánk a megoldást.

### Interfész kód

```
class BallIfc {};  
class Ball : public BallIfc {};  
  
template<Ifc>  
class Port { void send(Ifc& e); };  
  
Port<BallIfc>
```

# Példa: Interfész kódgenerálási stratégiák

Az interfész legyen általános osztály annyi darab művelettel, ahány fogadó végpontja van. A port osztály valósítsa meg az interfészt, Hátrányok:

- Jelentősen megnő a generált kód mérete.
- Funkcionálisan az összes művelet ugyanazt csinálja (így sablon metaprogramozással egyszerűsíthető..).

## Interfész kód

```
class BallIfc {  
public:  
virtual void send(Ball& e) { sendAny(e)};  
};
```

```
template<Ifc>  
class Port : public Ifc {  
protected:  
void sendAny(Event& e) {...}  
};
```



## Példa: Interfész kódgenerálási stratégiák

Az interfészt bontjuk két részre, hogy a belső üzenetküldést, és a kívülről jött üzenetfogadást is ki tudja fejezni.

### Interfész kód

```
class BallIfc {  
    class RequiredPart {  
        virtual void send(Ball& e) { sendAny(e); };  
    class ProvidedPart {  
        virtual void receive(Ball& e) { receiveAny(e); };  
    };  
};  
  
template<Provided, Required>  
class Port :  
    public Provided::ProvidedPart,  
    public Required::RequiredPart {  
  
};
```

- Komponens modellezés támogatása (környezetfüggetlenség, újrafelhasználhatóság, stb.).
- txtUML -> FMU exportálást hatékonyabban lehet megoldani.
- Oktatási célok.

- Az UML kompozit szabvány alapos értelmezése.
- Helyes reprezentáció használata.
- C++ kódgenerálási stratégiák elemzése, implementációval validálás.
- Szabványos modell generálása txtUML modell alapján, egy stratégia implementációja.
- *txtUML* támogatja komponens-alapú modellezést, felhasználása..

Köszönöm a figyelmet!