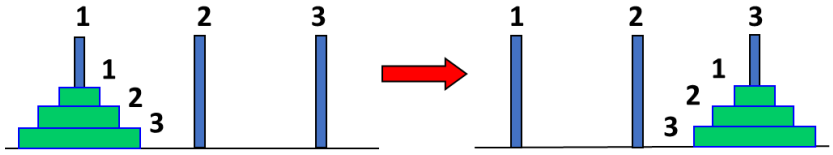
6. hét

Cél: Az alábbi alkalmazásban is élesen szétválik a modell a nézettől. Saját vezérlőket készítünk, és megismerkedünk a drag&drop technikával.

Hanoi tornyai probléma

Adott három rúd és néhány különböző méretű közepén lyukas korong. Kezdetben minden korong az egyik rúdon található úgy, hogy alulról felfelé haladva csökken a méretük, és ezáltal egy kúpszerű alakzatot formálnak. A játék során lehetőség van egy korongnak egy másik rúdra történő áthelyezésére, de természetesen az áthelyezendő korong felett más korong nem lehet, és ahová áttesszük, ott nem rakhatjuk nálánál kisebb korong tetejére. A cél az, hogy az összes korongot áttegyük egy másik rúdra a harmadik rúd segédletével, és a megoldás során az összes rúdon kúpszerű alakzatot kell a korongoknak felvenniük.



Érdemes a korongokat nagyságuk sorrendjében 1-től indulva megszámozni (a legkisebb korong az 1-es), valamint a rudakat is sorszámuk alapján megkülönböztetni.

Modellezés

A probléma állapottere tartalmazza a korongok összes lehetséges elhelyezkedését leíró állapotokat. Egyetlen állapot leírható (ennek megfelelően reprezentálható) egy olyan tömbbel, amelynek hossza a korongok száma, az *i*-dik indexű értéke pedig azt a rudat (rúd sorszámot) mutatja, ahol az *i*-dik korong az adott állapotban éppen megtalálható. A fenti ábrán (ahol három korong szerepel) a kezdőállapotot a [1, 1, 1] tömbbel, a célállapotot az [3, 3, 3] tömbbel jellemezhetjük.

Egy állapotból (*state*) egy másikba a korong-mozgatás műveletével juthatunk. Ennek a műveletnek elég két paramétert megadni: annak a rúdnak a számát ahonnan a legfelső korongot át akarjuk helyezni, és annak a rúdnak a számát, ahová a korongot rakni akarjuk.

MoveDisk**(from, onto):**AT→AT

HA a *from* és *onto* rudak léteznek és nem azonosak, továbbá van korong a *from* rúdon, és az *onto* rúd vagy üres, vagy a legfelső korongja nagyobb, mint a *from* rúd legfelső (mozgatandó) korongja

AKKOR *state*[*from* legfelső korongja] *:= onto*

Mindezek alapján elkészíthetjük a Hanoi tornyai probléma osztályát. Ennek egy objektuma egy állapotot ír le, rendelkezik a mozgatás (*move()*) művelete mellett a kezdőállapotot beállító (*init()*), és a célállapotot felismerő (*final()*) művelettel. A korongok számát az *init()* álltja be.

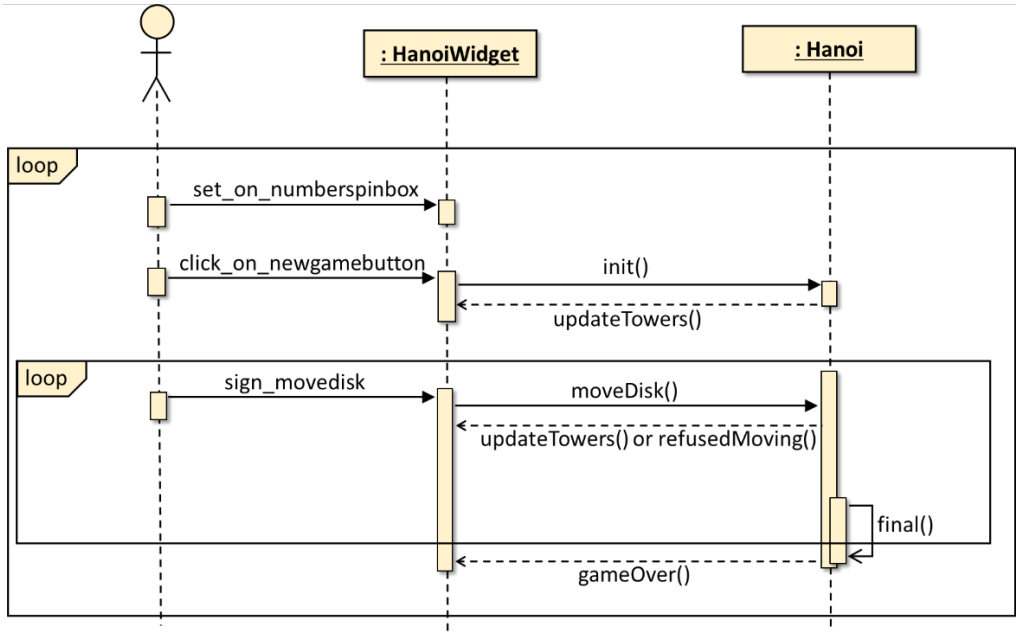
Architektúra

A modell-nézet architektúrában külön osztályokkal valósítjuk meg a modellt és nézetet. Miután a nézet példányosítja és aggregálja a modellt, fogadja és feldolgozza a felhasználó és a modell által generált eseményeket.

A felhasználó egy számbeállítón megadja a korongok számát (*set_on_numberspinbox*), majd egy nyomógombbal elindítja a játékot (*click_on_newgamebutton*). Ezután (számbeállítók és egy nyomógomb segítségével, vagy drag&drop technikával) jelzi, hogy melyik rúdról melyik rúdra akar korongot mozgatni (*on_movedisk*).

A felhasználó jelzéseire a nézet a modell megfelelő metódusait hívja.

A modell jelez a nézetnek, ha frissíteni kell a képet (*updateTowers()*), vagy ha egy lépés nem végrehajtható (*refusedMoving()*), illetve ha sikerült megoldani a problémát (*gameOver()*).



A nézethez két változatot is készítünk. Az elsőben egy korong mozgatásának paramétereit (honnan, hová) külön vezérlőkkel kell majd megadni, a másodikban drag&drop technikát használunk.

1. „Fapados" megoldás

Projekt létrehozása

Hozzunk létre egy *Qt Widget Application* projektet. Egy modell-nézet architektúrájú alkalmazást készítünk, ahol a főablakunk a *HanoiWidget* osztály példánya lesz, amelynek ősosztálya *QWidget*. A főablak felületének van statikus rész, amelyet a *QtDesigner* segítségével tervezzünk meg, a korongok aktuális elhelyezkedésének megjelenítését a kóddal kell implementálnunk.

Modell

A *Hanoi* osztály tartalma már a modellezés során kialakult. Figyelembe véve a C++ sajátosságait, a rudakat is, és a korongokat is 0-val kezdődően fogjuk sorszámozni.

Hanoi
- state : int[] - size : int
+ init() : void + final() : bool + moveDisk(from:int, onto:int) : bool <<getter>> + getDiskNumber() : int + getDisksOnTower(int) : int[]

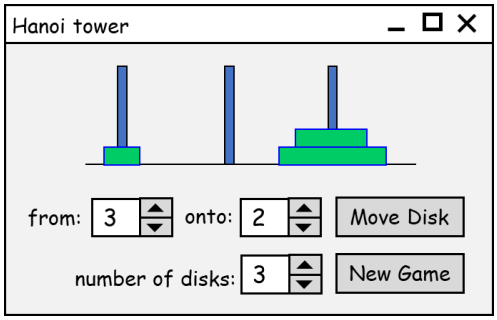
Megjelenik az osztályban két getter, amelyekre a nézetnek lesz szüksége. A *getDiskNumber()* adja vissza a korongok számát, de ezt a C++ megvalósításban kell külön *size* adattagban tárolni, hiszen ez a *state*-től lekérdezhető. A *getDisksOnTower()* egy adott rúdon elhelyezkedő korongok sorszámainak sorozatát adja vissza alulról felfele (azaz a korongok mérete szerint csökkenő sorrendben).

Az *init()* beállítja a start állapotot a *state*-ben. A *final()* ellenőrzi, hogy a *state* nem a célállapot-e. A *moveDisk()* a modellezés során bevezetett műveletet valósítja meg. Ez a metódus hamis értéket ad vissza, ha a művelet meghiúsult, és igaz értéket, ha megváltoztatja az aktuális állapotot (*state*).

Hozzáadunk az osztályhoz még három szignált:

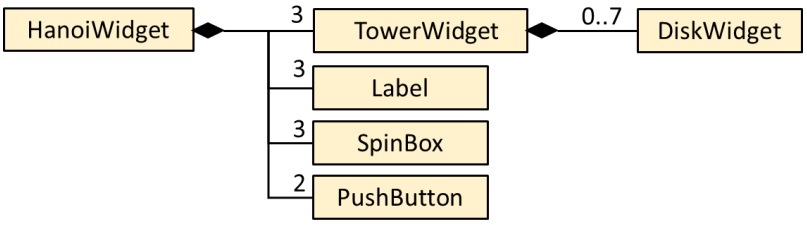
- az *updateTowers()*-t akkor váltjuk ki, amikor az állapot változik: egyrészt az *init()* végén, másrészt a *moveDisk()*-ben, de csak akkor, ha a lépés sikeres;
- a *movingRefused()*-t akkor váltjuk ki a *moveDisk()*-ben, ha a lépés sikertelen;
- a *gameOver()*-t is a *moveDisk()*-ben váltjuk ki, ha sikeres lépés után az aktuális állapotra *final()* igazat ad.

Felhasználói felület



A felhasználói felülethez kell

- három speciális vezérlő (*TowerWidget*), amellyel a tornyokat ábrázoljuk,.
- a tornyokat alkotó korongok (egy torony legfeljebb hét korongból állhat) megjelenítésére szolgáló speciális vezérlők (*DiskWidget*),
- két nyomógombra: egy korong mozgatásához, az új játék indításához,
- három számbeállítóra: a korong-mozgatás rúdjainak megadásához, a korongok számának beállításához,
- néhány címkére, és elrendezőkre.



DiskWidget

Egy korong megjelenítésére szolgál a *DiskWidget* osztály (őse a *QWidget*). Ennek két statikus adattagja lesz: a korong magassága (*diskHeight*), és a maximális szélessége (*maxDiskWidth*) pixelben (pl. 300x50).

Az osztály konstruktora megkapja az adott korong sorszámát, és a játékban résztvevő korongok számát. Rögzíti a korong méretét (*setFixedSize()*): a korong szélessége a maximális méretből a korong sorszámának és a korongok összes számának hányadosával számítható ki. A korong háttérszínét egy *QPalette* objektum írja le: a szín is a korong méretétől függ. Végül megjelenítjük a widget-et a show() metódussal.

```
DiskWidget::DiskWidget(int diskSize, int diskNumber, QWidget* parent)
   : QWidget(parent)
{
   setFixedSize(maxDiskWidth * (diskSize+1) / diskNumber, diskHeight);
   QPalette pal = palette();
   const int colorValue = 255 * (diskNumber-diskSize-1) / (diskNumber-1);
   pal.setColor(QPalette::Background, QColor(colorValue, 0, 0));
   setAutoFillBackground(true);
   setPalette(pal);
   show();
}
```

TowerWidget

Egyetlen torony (néhány korong egymás tetején) megjelenítéséhez vezetjük be a *TowerWidget* osztályt (őse a *QWidget*). Ennek az osztálynak három példányára lesz majd szükségünk, mivel három rúd szerepel a játékban. Egy torony rendelkezik a rúdjának a sorszámával (*_id*), és ismeri a modellt (ezeket a konstruktorán keresztül kapja meg). A torony megjelenítéséhez egy *QBoxLayout* típusú elrendezőt (*_layout*) használunk, amelybe különböző méretű és színű *DiskWidget*-et (azaz korongot) fogunk elhelyezni.

```
TowerWidget::TowerWidget(Hanoi* m, int id, QWidget* parent)
   : QWidget(parent), _model(m), _id(id)
{
   _layout = new QBoxLayout(QBoxLayout::BottomToTop, this);
   _layout->setSpacing(0);
}
```

A tornyot alkotó korongok kirajzolását az *on_updateDisks()* eseménykezelő függvény végzi, amelyet a modell aktivál az updateTowers() szignál kiváltásával. A rajzoláshoz minden fontos információt a modelltől kérdezhetünk le.

```
void TowerWidget::on_updateDisks()
{
   while (_layout->count() != 0) {
      QLayoutItem* disk = _layout->takeAt(0);
      delete disk->widget();
   }

   const Disks& disks = _model->getDisksOnTower(_id);

   for (int i = 0; i < disks.size(); ++i) {
      DiskWidget* disk =
         new DiskWidget(disks[i], _model->getDiskNumber(), this);
      _layout->addWidget(disk, 0, Qt::AlignHCenter | Qt::AlignBottom);
   }
   _layout->addStretch(_model->getDiskNumber() - disks.size());
}
```

Először a torony korábbi állapot szerinti korongjait (*DiskWidget*-jeit) kell törölnünk.

Ezután lekérjük a modelltől a toronyban az aktuális állapot szerint jelenlevő korongok sorszámait csökkenő sorrendben tartalmazó sorozatot, majd létrehozzuk a korongok megjelenítésére szolgáló *DiskWidget*-eket, és elhelyezzük őket a tornyot jelképező *_layout* elrendezőben (mindig alulra és középre). A korongok feletti helyet is töltsük ki (*addStretch()*). Ennek hiányában a korongok nem közvetlenül egymáson, hanem széthúzódva jelennének meg.

HanoiWidget eseménykezelése

A nézet főablaka négy eseménykezelő metódust vezet be.

```
private:
   Ui::HanoiWidget *ui;
   Hanoi* _model;
private slots:
   void on_newGame();
   void on_moveDisk();
   void on_movingRefused();
   void on_gameOver();
```

Az eseménykezelőknek a megfelelő eseményekhez kötését a konstruktor végzi. Itt kerül sor a modell példányosítására, valamint a tornyok (*TowerWidget*-ek) létrehozására. A tornyoknak az aktuális állapot szerinti kirajzolását az *updateTowers* szignál váltja ki.

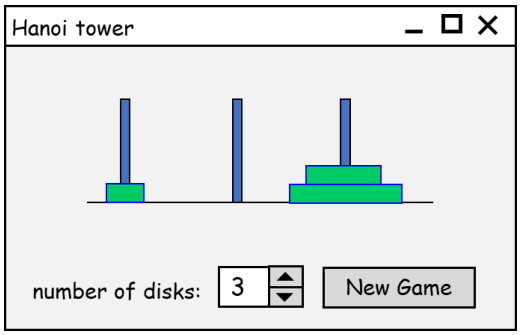
```
HanoiWidget::HanoiWidget(QWidget *parent)
   : QWidget(parent) , ui(new Ui::HanoiTowerWidget)
{
   ui->setupUi(this);
   setMinimumWidth((DiskWidget::maxDiskWidth + 20) * Hanoi::towerCount);

   _model = new Hanoi();

   for (int i = 0; i < Hanoi::towerCount; ++i) {
      ui->gridLayout->setColumnStretch(i, 1);
      TowerWidget* tower = new TowerWidget(_model, i, this);
      ui->gridLayout->addWidget(tower, 0, i);
      connect(_model, SIGNAL(updateTowers()), tower, SLOT(on_updateDisks()));
   }
   connect(ui->newGameButton, SIGNAL(clicked()),this, SLOT(on_newGame()));
   connect(ui->moveDiskButton,SIGNAL(clicked()),this,SLOT(on_moveDisk()));
   connect(_model, SIGNAL(movingRefused()), this, SLOT(on_movingRefused()));
   connect(_model, SIGNAL(gameOver()),      this, SLOT(on_gameOver()));
   on_newGame();
}
```

Az *on_newGame()* a modell *init()* metódusát, az *on_moveDisk()* a modell *moveDisk()* metódusát hívja. Az *on_Refused()* és az *on_gameOver()* egy-egy megfelelő tartalmú szöveget jelenít meg egy üzenet-ablakban. Mindhárom torony *on_updateDisks()* eseménykezelőjét a modellben kiváltódó *updateTowers()* szignálhoz kötjük.

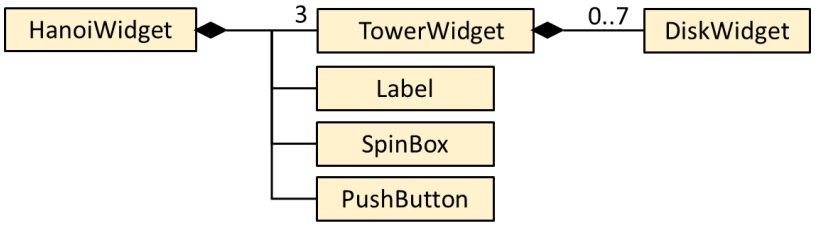
1. Egyszerű drag&drop megoldás

Az előző projektet módosítjuk. Egy korongot úgy szeretnénk mozgatni egyik toronyból a másikba, hogy az egér segítségével megragadunk az egyik tornyok tetején egy korongot, és azt átvonszoljuk (drag) egy másik toronyra, ahol leejtjük (drop).

Felhasználói felület



A felhasználói felületről lekerülnek tehát a korong-mozgatás megvalósításához korábban használt vezérlők.



HanoiWidget

A felület egyszerűsödése következtében ki kell törölnünk a *HanoiWidget* osztályból az *on_MoveDisk()* eseménykezelőt. A modell *moveDisk()* metódusát majd a drag&drop tevékenység következtében fogjuk meghívni.

DiskWidget

A *DiskWidget* osztályban felüldefiniáljuk a *mousePressEvent()* metódust. Ezt az váltja ki, ha a vezérlő fölé mozgatva az egérkurzort megnyomjuk valamelyik egérgombot.

Csak a bal egérgombbal engedjük meg a vonszolást, azért először ezt ellenőrizzük. Ezután hozzunk létre egy vonszolás (drag) objektumot, amellyel elindítjuk a drag&drop folyamatot. (Erre a folyamatra utal, hogy megváltozik az egérkurzor alakja.) A vonszolás objektum *mimeData* tagja a vonszolás tárgyának leírására szolgál. Mi ennek segítségével adjuk meg a megragadott korongot tartalmazó tornyot (szülő vezérlő), hogy majd amikor befejeződik a drag&drop folyamat (mert felengedjük az egérfület, azaz elejtjük a vonszolt korongot), akkor álljon majd rendelkezésünkre ez az információ.

```
void DiskWidget::mousePressEvent(QMouseEvent* event) {
    if (event->button() == Qt::LeftButton) {
        QDrag* drag = new QDrag(parentWidget());
        QMimeData* mimeData = new QMimeData();
        mimeData->setParent(parentWidget());
        drag->setMimeData(mimeData);
        drag->exec();
    }
}
```

TowerWidget

A legtöbb változtatást a *TowerWidget* osztályán kell elvégeznünk.

Kiegészítjük a konstruktort egy új utasítással. A *setAcceptDrops(true)* engedélyezi, hogy a toronyvezérlő felületén elejthessünk egy vonszolt objektumot. Ez váltja li az elejtés eseményt.

Felüldefiniáljuk a *QWidget*-től örökölt *dragEnterEvent()*, és *dropEvent()* eseménykezelőket. A *dragEnterEvent()* akkor fut le, amikor a drag&drop tevékenység során egy vezérlő fölé ér az egérkurzor. Ha az esemény által vonszolt objektum a szülője eredetileg is egy torony-vezérlő volt (ezt az eseményhez beállított *QMimeData* típusú adattag *parent* mezője mutatja meg), akkor elfogadjuk az eseményt az *acceptProposedAction()* meghívásával, hiszen elejtés esetén le fogjuk tudni kezelni azt.

```
void TowerWidget::dragEnterEvent(QDragEnterEvent* event) {
   const TowerWidget* parentTower =
            qobject_cast<TowerWidget*>(event->mimeData()->parent());
   if (parentTower) event->acceptProposedAction();
}
```

A *dropEvent()* akkor fut le, amikor egy drag&drop tevékenység során elejtjük a vonszolt objektumot. Előbb meg kell győződnünk arról, hogy a vonszolt objektum egy korong-e, azaz eredetileg egy torony volt a szülője, és felhasználva a küldő és a fogadó torony sorszámát (ehhez bevezetünk egy gettert) meghívjuk a modell egy lépését (*move Disk()*). Ha a lépés legális, akkor elfogadjuk a drag&drop tevékenységet (*event->accept()*), ha nem, akkor figyelmen kívül hagyjuk (*event->ignore()*).

```
void TowerWidget::dropEvent(QDropEvent* event) {
   const TowerWidget* parentTower =
            qobject_cast<TowerWidget*>(event->mimeData()->parent());
   if (parentTower) {
      if (_model->moveDisk(parentTower->getId(), _id)) event->accept();
      else event->ignore();
   }
}
```

Az *on_updateTowers()* eseménykezelőn annyit kell módosítani, hogy kikötjük, hogy egy toronyban megjelenő korong-vezérlők közül csak a legfelsőt lehessen megragadni egy drag&drop folyamat során. Ehhez a korongokat megjelenítő *DiskWidget*-ekre (azok létrehozása után) a *setEnable()* metódust kell meghívni *true* vagy *false* paraméterrel.

2. Haladó drag&drop megoldás

Javítsunk a drag&drop technikán. Érjük el, hogy egy korong a vonszolás során folyamatosan az egérkurzort követve mozogjon. Ennek megvalósításához

- a drag&drop folyamat elején megragadott korong törlődjön a küldő torony vezérlőjéből,
- a megragadott korong az egérkurzorral együtt mozogjon,
- a fogadó tornyot kiválasztva, és a korongot ott elejtve – feltéve, hogy ez a lépés szabályos – a leejtett korong kerüljön rá a céltoronyra,
- ha a lépés nem legális, akkor rendeződjön vissza a drag&drop folyamat előtt állapot.

DiskWidget

Érdemes a *mousePressEvent()* eseménykezelő mellett a *mouseMoveEvent()* eseménykezelőt is felüldefiniálni. A *mousePressEvent()* csak éppen, hogy elindítja a drag&drop folyamatot azzal, hogy ha bal egérgombot lenyomjuk megjegyzi az egérkurzor pozícióját.

```
void DiskWidget::mousePressEvent(QMouseEvent* event)
{
   if (event->button() == Qt::LeftButton) {
      _movingStart = false;
      _startPos = event->pos();
   }
}
```

A *mouseMoveEvent()* megnézi, hogy az egérkurzus határozottan elmozdult-e a bal egérgomb lenyomása mellett.

```
void DiskWidget::mouseMoveEvent(QMouseEvent* event)
{
   if (event->buttons() & Qt::LeftButton) {
      int distance = (event->pos() - _startPos).manhattanLength();
      if(!_movingStart && distance >= QApplication::startDragDistance()) {
         _movingStart = true;
```

Amikor ezt először érzékeljük (ennek figyelésére való a *DiskWidget*-nek a *_movingStart* adattagja), akkor végrehajtjuk az előző megoldásban még a *mousePressEvent()* eseménykezelőben szerepeltetett tevékenységeket.

```
         QDrag* drag = new QDrag(parentWidget());
         QMimeData* mimeData = new QMimeData();
         mimeData->setParent(parentWidget());
         drag->setMimeData(mimeData);
```

Könnyen megvalósítható, hogy az egérkurzor képe a korongot jelképező (megfelelő színű) téglalap legyen. Ehhez a korong színét (*_color*) a *DiskWidget* adattagjaként eltároljuk, és a konstruktorban beállítjuk (*_color = QColor(colorValue, 0, 0)*). Így a vonszolás során a megragadott korong képe fog mozogni.

```
            QPixmap cursor(width(), height());
            cursor.fill(_color);
            drag->setDragCursor(cursor, Qt::DropAction::MoveAction);
```

El kell távolítani a megragadott korongot a küldő torony tetejéről.

```
            TowerWidget* parentTower = qobject_cast<TowerWidget*>(parent());
            if (parentTower->_layout->count() != 0) {
                QLayoutItem* disk =
                parentTower->_layout->takeAt(parentTower->_layout->count()-2);
                delete disk->widget();
            }
            drag->exec(Qt::MoveAction);
        }
    }
}
```

Két problémát kellene még megoldanunk:

1. Ha a drag&drop tevékenység nem egy *TowerWidget* felett fejeződik be (ilyenkor az egérkurzor ideiglenesen megváltozik), akkor a leejtett korong visszakerül a küldő rúdra, de ez nem látszik, mert nem frissül a kép, így a leejtett korong ideiglenesen eltűnik.
2. Kis mértékben változik a tornyok szélessége korongmozgatás során, és ez zavaró.
   `tower->setSizePolicy(QSizePolicy::Ignored, QSizePolicy::Ignored)`