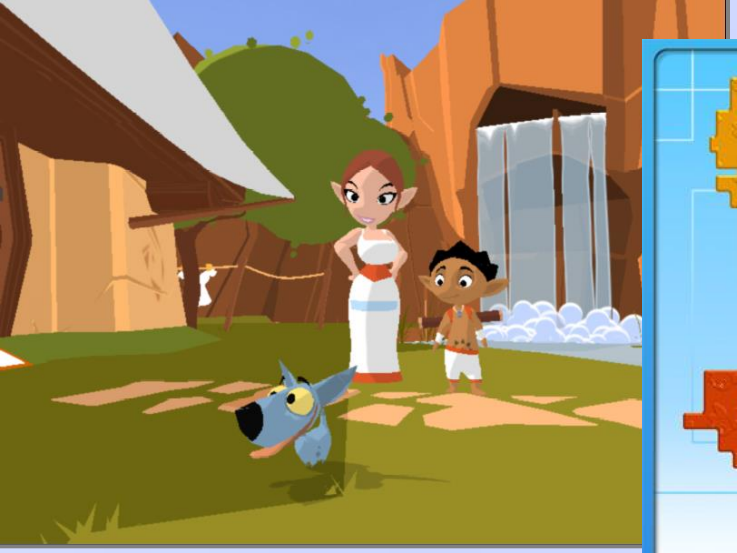
A simple and powerful property system for C++

GCDC 2008

David Salz
david.salz@bitfield.de



- **Bitfield**
  - small studio from Berlin, Germany
  - founded in 2006
  - developer for PC, DS, Wii

  - focus on casual games for PC and DS
  - advertising games for trade shows & events

GIC
DEVELOPER CONFERENCE

- **Bitfield**
  - licensed DS middleware developer
  - BitEngineDS, high level game engine
    - used in 10 games so far
    - adventure, jump‘n‘run, puzzle games, pet simulation
  - www.bitfield.de

BITENGINE

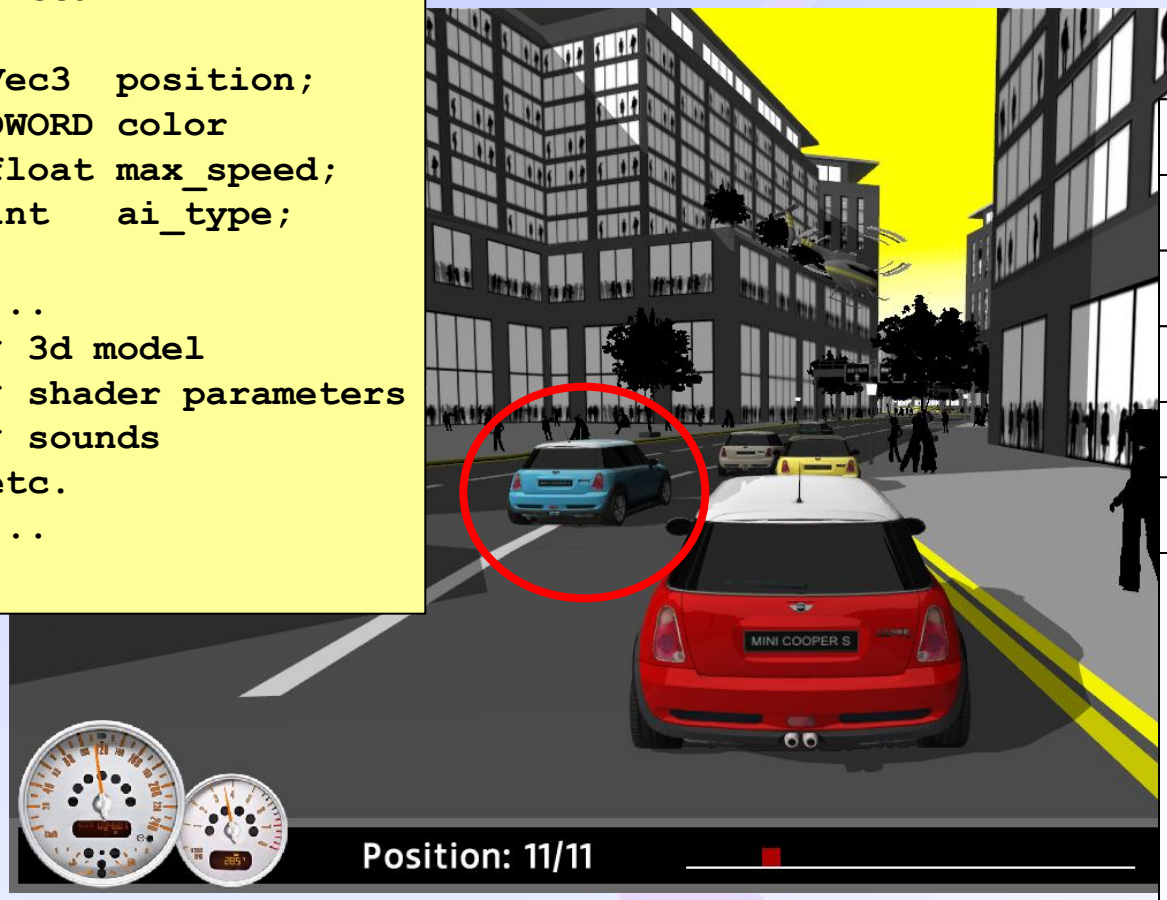
- Games Academy
  - founded in 2000
  - based in Berlin and Frankfurt/Main
  - ~170 students

- Hire our students!
- Constantly looking for teachers and lecturers!

What is a Property System?

```
class CCar
{
    Vec3  position;
    DWORD color
    float max_speed;
    int   ai_type;

    ...
    * 3d model
    * shader parameters
    * sounds
    etc.
    ...
};
```



location	127.3, 0, 14.0
max speed	120 km/h
color	...
AI Type	Easy ▼
Team	2 ▲▼
3d model	mini.x ...

Uses of Property Systems (1)

- a property system can be used for a lot of things!

- **serialization**
  - store objects on disc (or stream over network)
  - e.g. saved games, level file format, scene graph file format

- **undo / redo**
  - figure out which properties were changed, store old value
  - = selective serialization

- **client / server synchronization**
  - send changed properties to client/server
  - = selective serialization

Uses of Property Systems (2)

- a property system can be used for a lot of things!

- **animation**
  - change property values over time
  - e.g. animated GUIs, in-game cut-scenes

- **integration of scripting languages**
  - automatically create „glue code“ to access object properties from scripting languages

What do we want? (1)

- **easy-to-use system**
- **lightweight**
  - cause as little performance and memory overhead as possible
  - performance may not be an issue for editors...
  - ...but may for serialization, animation, scripting languages
- **type safe**
  - i.e. all C++ type safety and type conversion mechanisms should stay intact
- **non-intrusive**
  - i.e. it should not require modification of the class whose properties are to be exposed
  - it might by part of an external library; source code may not be available

What do we want? (2)

- **support polymorphic objects**
  - i.e. work correctly with virtual functions
- **be aware of C++ protection mechanisms**
  - i.e. not violate const, private or protected modifiers
- **support real-time manipulation of objects with no extra code**
  - i.e. the object should not need extra code to deal with a changed property at runtime

Existing Solutions

- ... using **data pointers**
- ... using **strings**
- … using **databases**
- ... using **reflection**
- **properties in Microsoft .NET**

- ... using **member function pointers**

Data Pointers (1)

- „A Property Class for Generic C++ Member Access“
  - by Charles Cafrelli, published in *Game Programming Gems 2*
- Idea:
  - store a pointer to the data (inside to object)
  - plus type information
  - plus name

Data Pointers (2)

```
class Property
{
protected:
   union Data
   {
      int* m_int;
      float* m_float;
      string* m_string;
      bool* m_bool;
   };

   enum Type
   {
      INT,
      FLOAT,
      STRING,
      BOOL,
      EMPTY
   };

   Data m_data;
   Type m_type;
   string m_name;
```

```
   Property(string const& name, int* value);
   Property(string const& name, float* value);
   Property(string const& name, string* value);
   Property(string const& name, bool* value);
   ~Property ();

   bool SetUnknownValue(string const& value);
   bool Set(int value);
   bool Set(float value);
   bool Set(string const& value);
   bool Set(bool value);

   void SetName(string const& name);
   string GetName() const;

   int GetInt();
   float GetFloatf);
   string GetString();
   bool GetBool();
}
```

Data Pointers (3)

```
class PropertySet
{
protected:
  HashTable<Property>m_properties;
public:
  PropertySet();
  virtual -PropertySet();

  void Register(string const& name, int* value);
  void Register(string const& name, float* value);
  void Register(string const& name, string* value);
  void Register(string const& name, bool* value);

  // look up a property
  Property* Lookup(string const& name);

  // get a list of available properties
  bool SetValue(string const& name, string* value);
  bool Set(string const& name, string const& value);
  bool Set(string const& name, int value);
  bool Set(string const& name, float value);
  bool Set(string const& name, bool value);
  bool Set(string const& name, char* value);
};
```

```
class GameObj : public PropertySet
{
   int m_test;

   GameObj()
   {
       Register("test",&m_test);
   }
};
```

```
Property* testprop =
    aGameObj->Lookup("test");

int test_value =
    testprop->GetInt();
```

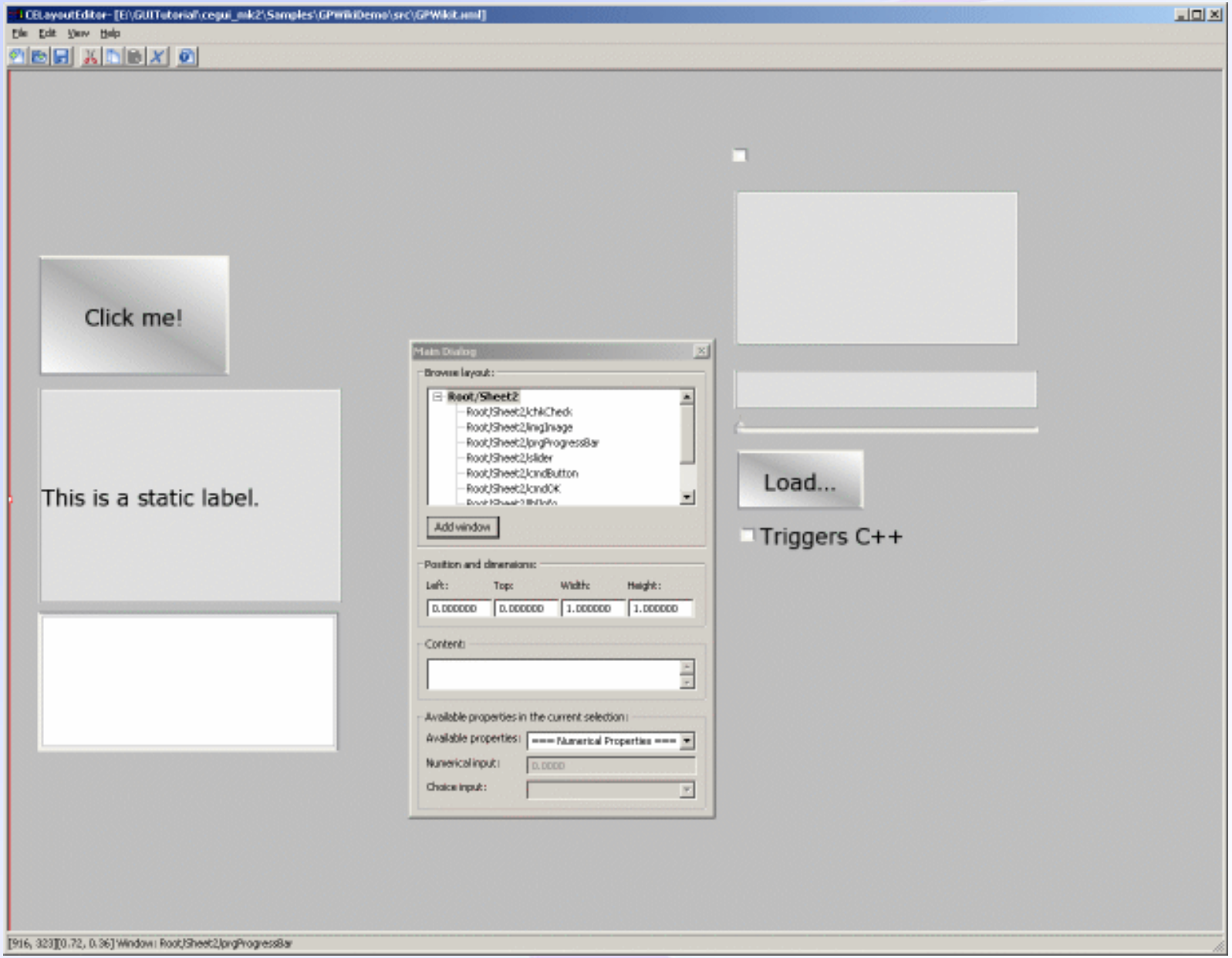
Data Pointers (4)

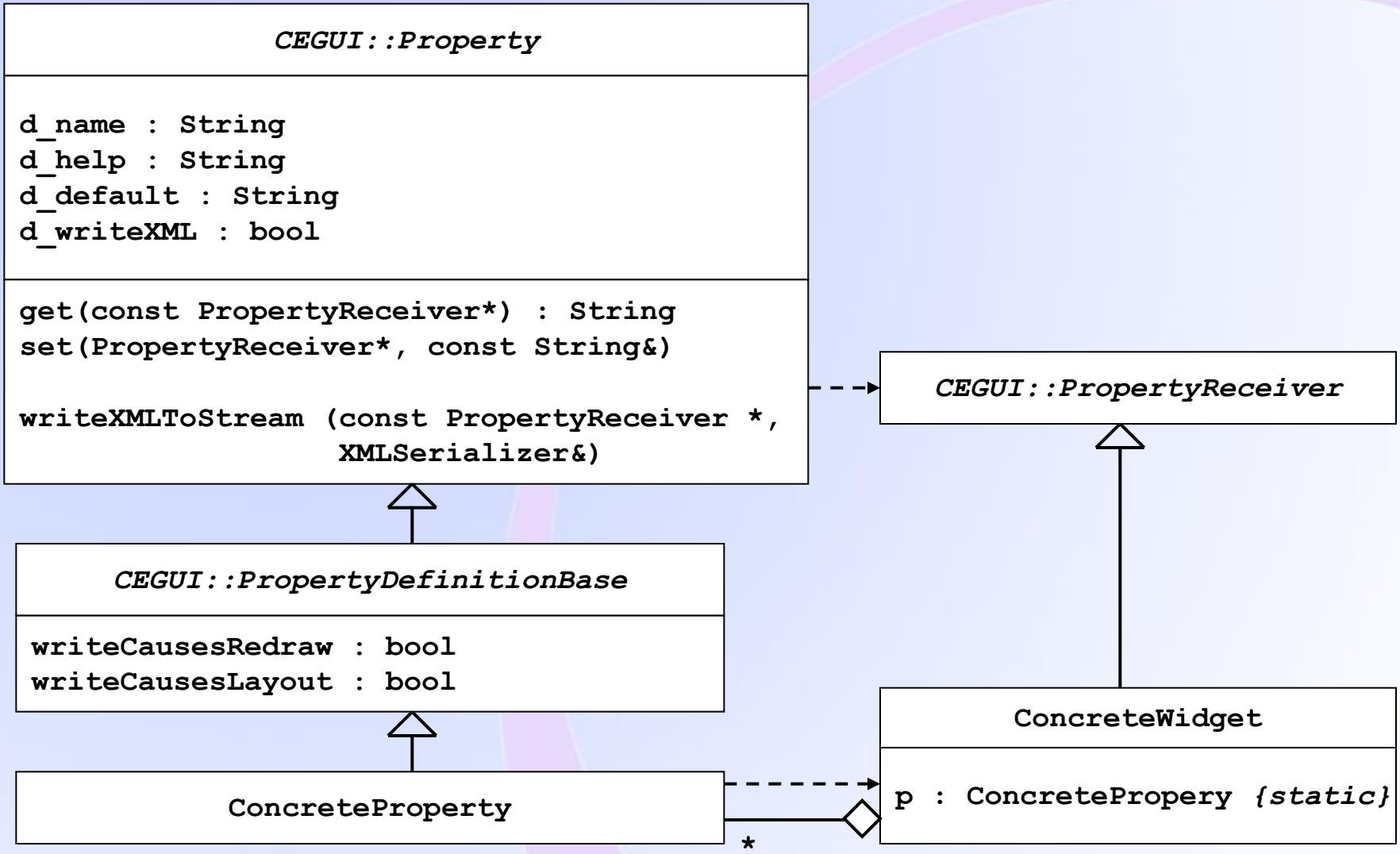
- Problems:
  - Is it a good idea to modify a (private?) variable directly?
  - → need to notify object after property change!
  - objects carry unnecessary and redundant data
    - every GameObj holds string names for all properties...
    - ... and pointers to it's own members!
  - support for more types?
    - could be done using templates
    - or by storing data as string internally

String-Based Properties

- Idea:
  - store all properties as strings
  - convert from / to string as needed

- used by CEGui
  - Crazy Eddie's GUI System; open source GUI
  - used with OGRE / Irrlicht game engines
  - used in several commercial games


CELayoutEditor - [E:\GUITutorial\cegui_mk2\Samples\GPWikiDemo\src\GPWiki.xml]
File Edit View Help
Click me!
This is a static label.
Main Dialog
Browse layout:
Root/Sheet2
Root/Sheet2/chkChecked
Root/Sheet2/img1mage
Root/Sheet2/prgProgressBar
Root/Sheet2/slider
Root/Sheet2/cmdButton
Root/Sheet2/cmdOK
Add window
Position and dimensions:
Left:
Top:
Width:
Height:
0.000000
0.000000
1.000000
1.000000
Content:
Available properties in the current selection:
Available properties:
==== Numerical Properties ====
Numerical input:
0.0000
Choice input:
Load...
Triggers C++
[916, 323][0.72, 0.36] Window: Root/Sheet2/prgProgressBar



- PropertyReceiver is just an empty class, used for type safety
- there is a derived class for every property of every class (181 !)

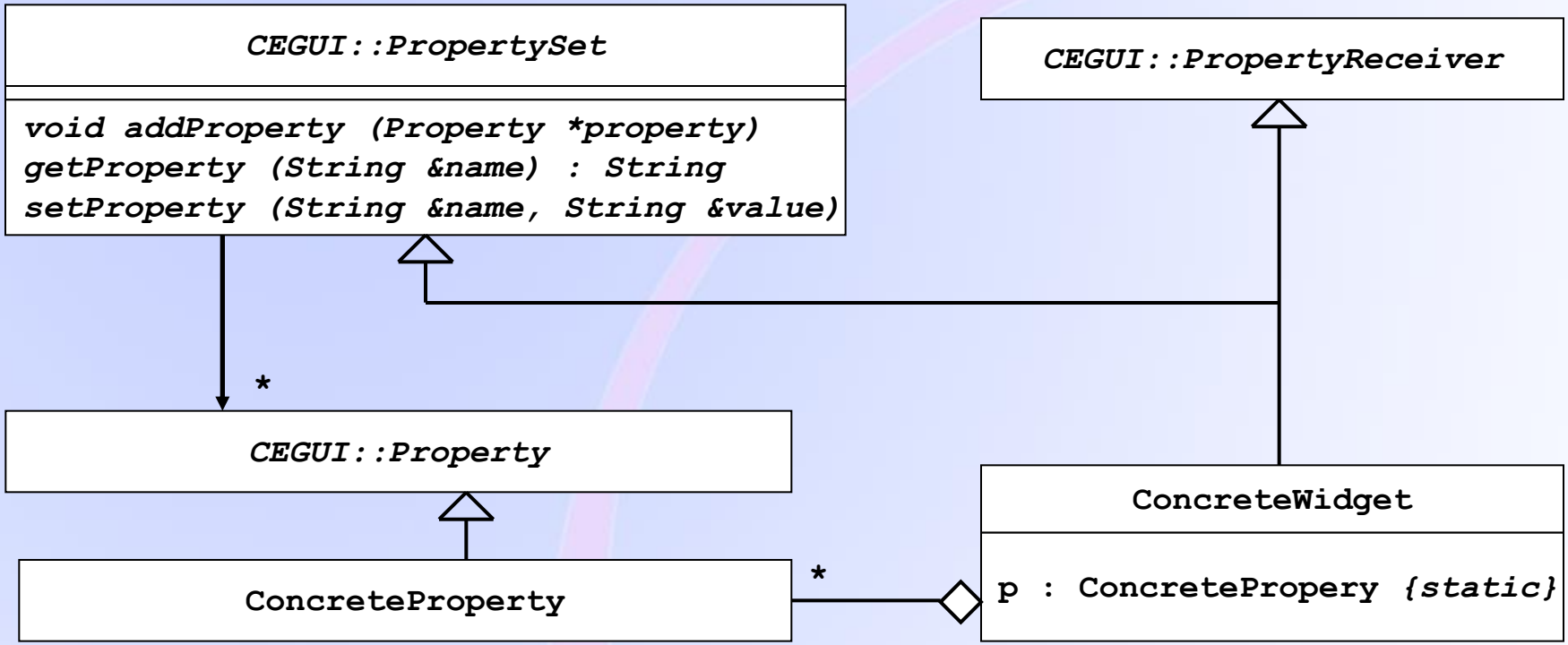
```
class Selected : public Property
{
public:

    Selected() : Property("Selected", "Property to get/set the selected state
      of the Checkbox. Value is either \"True\" or \"False\".", "False") {}

    String get(const PropertyReceiver* receiver) const
    {
        return PropertyHelper::boolToString(
               static_cast<const Checkbox*>(receiver)->isSelected());
    }

    void set(PropertyReceiver* receiver, const String& value)
    {
        static_cast<Checkbox*>(receiver)->setSelected(
            PropertyHelper::stringToBool(value));
    }
}
```

- concrete property...
  - ... converts string from / to correct type
  - ... casts receiver to correct target type
  - ... goes trough public API



- **PropertySet** holds a list of properties
- every Widget is a **PropertySet**; adds its own properties to the list

String-Based Properties

- Problems:
  - a lot of hand-written glue code
  - inefficient string conversions for every access
- Pros:
  - strings can be used for XML serialization
  - property data held per class, not per instance
  - goes through public interface of the class
    - class can react to changes properly

Databases

- Idea:
  - store all properties in a database (SQL / XML / whatever)
  - database API is single point of access for everything
  - obvious for Browser / Online games
  - Extreme: „There is no game entity class!“
- But:
  - There is always some code representation of the data
  - May not be feasable for small devices
  - Speed / memory footprint?

Reflection (1)

- **reflection**
  - ability of a program to observe its internal structure at runtime
    - assembly code in memory can be read and interpreted
    - some scripting languages treat source code as strings
  - typically on a high level
    - figure out type of an object at runtime (**introspection**; polymorphism)
    - get the class name as a string
    - get method names as strings
    - figure out parameters and return types of methods
  - supported by Objective C, Java, C#, VB .Net, Managed C++

Reflection (2)

- how does it work?
  - meta data generated by compiler
  - special interfaces to query type info from objects (= virtual functions)
  - lots of string lookups at runtime

C#

```
// trying to call:  String FooNamespace.Foo.bar()

Type t = Assembly.GetCallingAssembly().GetType("FooNamespace.Foo");
MethodInfo methodInfo = obj.GetType().GetMethod("bar");

// invoke the method, supplying parameter array with 0 size

value = (String)methodInfo.Invoke(obj, new Object[0]);
```

Reflection (3)

- Why doesn‘t C++ support this?
  - **no single-rooted hierarchy**
    - no common „Object“ base class for everything
    - void* is most “compatible” type
  - **primitive types not really integrated into OO structure**
    - Java and .NET have wrapper classes for them
    - .NET: “boxing” / “unboxing” mechanism
  - **C++ supports early and late binding**
    - i.e. supports non-virtual functions
    - ... and classed without a VTable
  - **RTTI works for objects with VTable only**
    - VTable is not generated just for RTTI

Reflection (4)

- Reflection Libraries for C++ (just examples)

- **Seal Reflex**
  - by European Organization for Nuclear Research (CERN)
  - uses external tool (based on) GCCXML) to parse C++ code and generate reflection info
  - non-intrusive; generates external dictionary for code
  - up to date; supports latest gcc and VisualC++
  - can add own meta-info to code (key-value-pairs)
  - http://seal-reflex.web.cern.ch

```
Type t = Type::ByName("MyClass");
void * v = t.Allocate(); t.Destruct(v);
Object o = t.Construct();

for (Member_Iterator mi = t.Member_Begin(); mi != t.Member_End(); ++mi)
{
    switch ((*mi).MemberType())
    {
    case DATAMEMBER    : cout << "Datamember: " << (*mi).Name() << endl;
    case FUNCTIONMEMBER : cout << "Functionmember: " << (*mi).Name() << endl;
    }
}
```

Reflection (5)

- Reflection Libraries for C++ (just examples)

- **CppReflection**
  - by Konstantin Knizhnik
  - reflection info supplied partly by programmer (through macros)
  - ... and partly taken from debug info generated by compiler
  - http://www.garret.ru/~knizhnik/cppreflection/docs/reflect.html

```
class A
{
public:
    int i;
protected:
    long larr[10];
    A** ppa;
public:
    RTTI_DESCRIBE_STRUCT((RTTI_FIELD(i, RTTI_FLD_PUBLIC),
                          RTTI_ARRAY(larr, RTTI_FLD_PROTECTED),
                          RTTI_PTR_TO_PTR(ppa, RTTI_FLD_PROTECTED)));
};
```

Reflection (5)

- **Boost Reflection**
  - by Jeremy Pack
  - not part of the boost libraries yet!
  - programmer must specify reflection info (through macros / templates)
  - http://boost-extension.blogspot.com
- also possible: **COM** and **CORBA**
  - APIs that provide communication across language boundaries
  - also contain type description facilities

.NET Property System (1)

C#

```
class CTestClass
{
    private int iSomeValue;
    public int SomeValue
    {
        get()   { return iSomeValue; }
        set()   { iSomeValue = value; }
    }
}
```

C#

```
CTestClass pTestClass = new CTestClass();

pTestClass.SomeValue = 42;                          // implicitly calls set()
System.Console.WriteLn(pTestClass.SomeValue);       // implicitly calls get()
```

- properties in .NET languages
  - ... look like member variables to the outside
  - ... but really are functions!
- → syntactic alternative to get/set functions

.NET Property System (2)

Managed C++

```
ref class CTestClass
{
private:
    int iSomeValue;

public:
    property int SomeValue
    {
        int get()              { return iSomeValue; }
        void set(int value)    { iSomeValue = value; }
    }
}
```

Managed C++

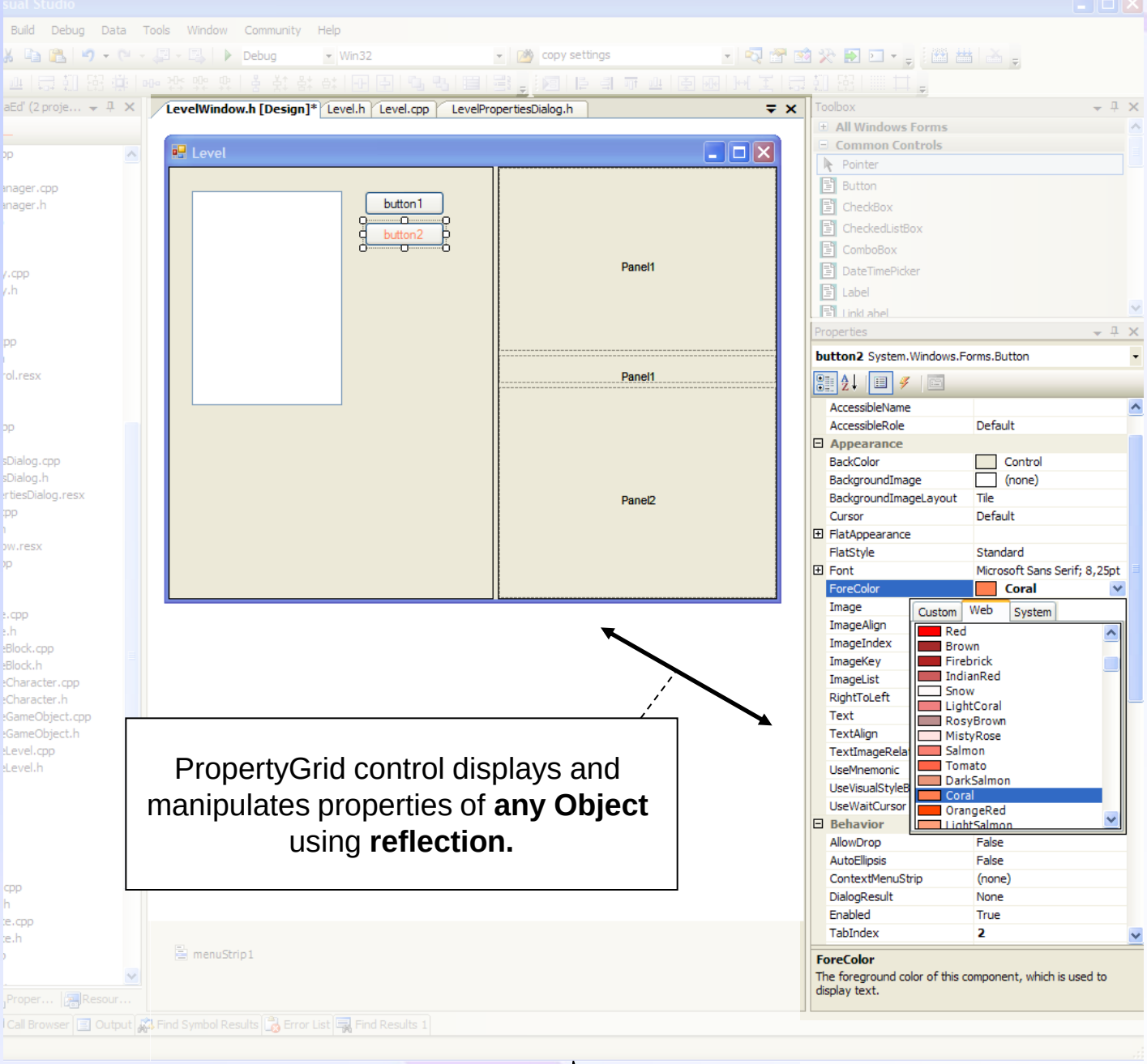
```
CTestClass^ pTestClass = gcnew CTestClass;

pTestClass->SomeValue = 42;
System::Console::WriteLn(pTestClass->SomeValue);
```

- just syntactic sugar?

.NET Property System (3)

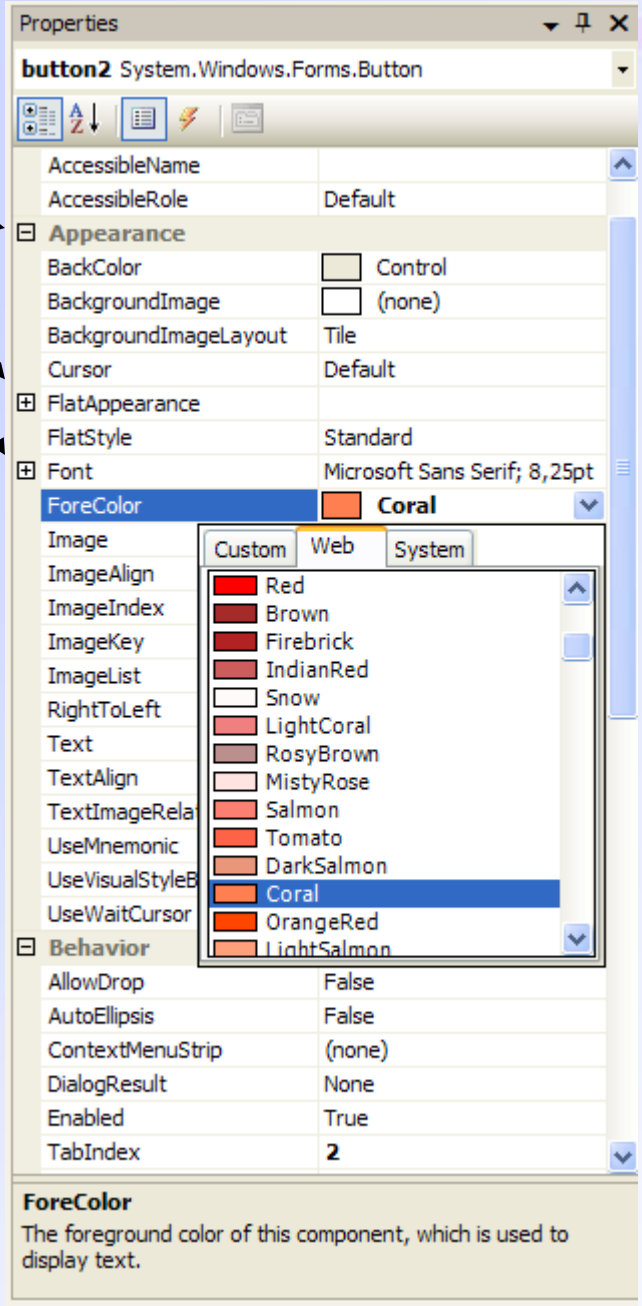
- **clean alternative to get/set methods**
  - look like one data field from the outside
  - can still allow only get or only set
  - can be abstract, inherited, overwritten, work with polymorphism
- **interesting in connection with reflection**
  - reflection can tell difference between a „property“ and normal methods
- **can add custom attributes to properties**
  - through System.Attribute class
  - e.g. description text, author name... anything you want


Level
button1
button2
Panel1
Panel1
Panel2
PropertyGrid control displays and manipulates properties of **any Object** using **reflection.**
button2 System.Windows.Forms.Button
ForeColor
Coral
Custom
Web
System
Red
Brown
Firebrick
IndianRed
Snow
LightCoral
RosyBrown
MistyRose
Salmon
Tomato
DarkSalmon
Coral
OrangeRed
LightSalmon
ForeColor
The foreground color of this component, which is used to display text.

named property **groups**

property **names**

description text for each property



custom editing controls
for different **property types**

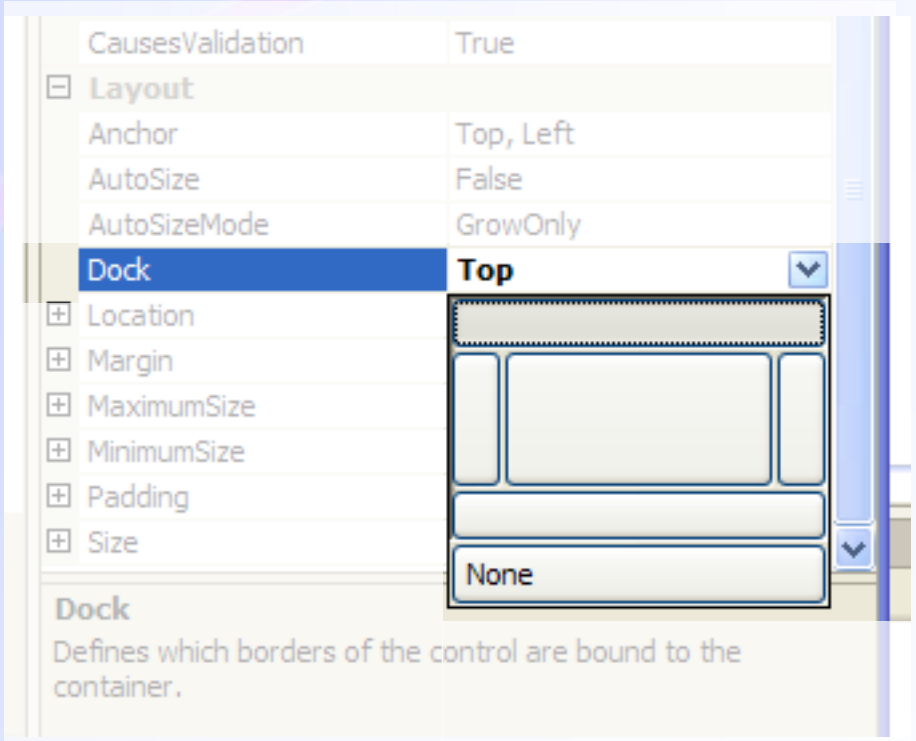
validity criteria,
e.g. minimum/maximum value,
list of valid values

lists of **predefined values**

(but some of that is domain knowledge the editor has and not part or the property system

```
public enum class DockStyle
{
    Bottom,
    Fill,
    Left,
    None,
    Right,
    Top
}
```

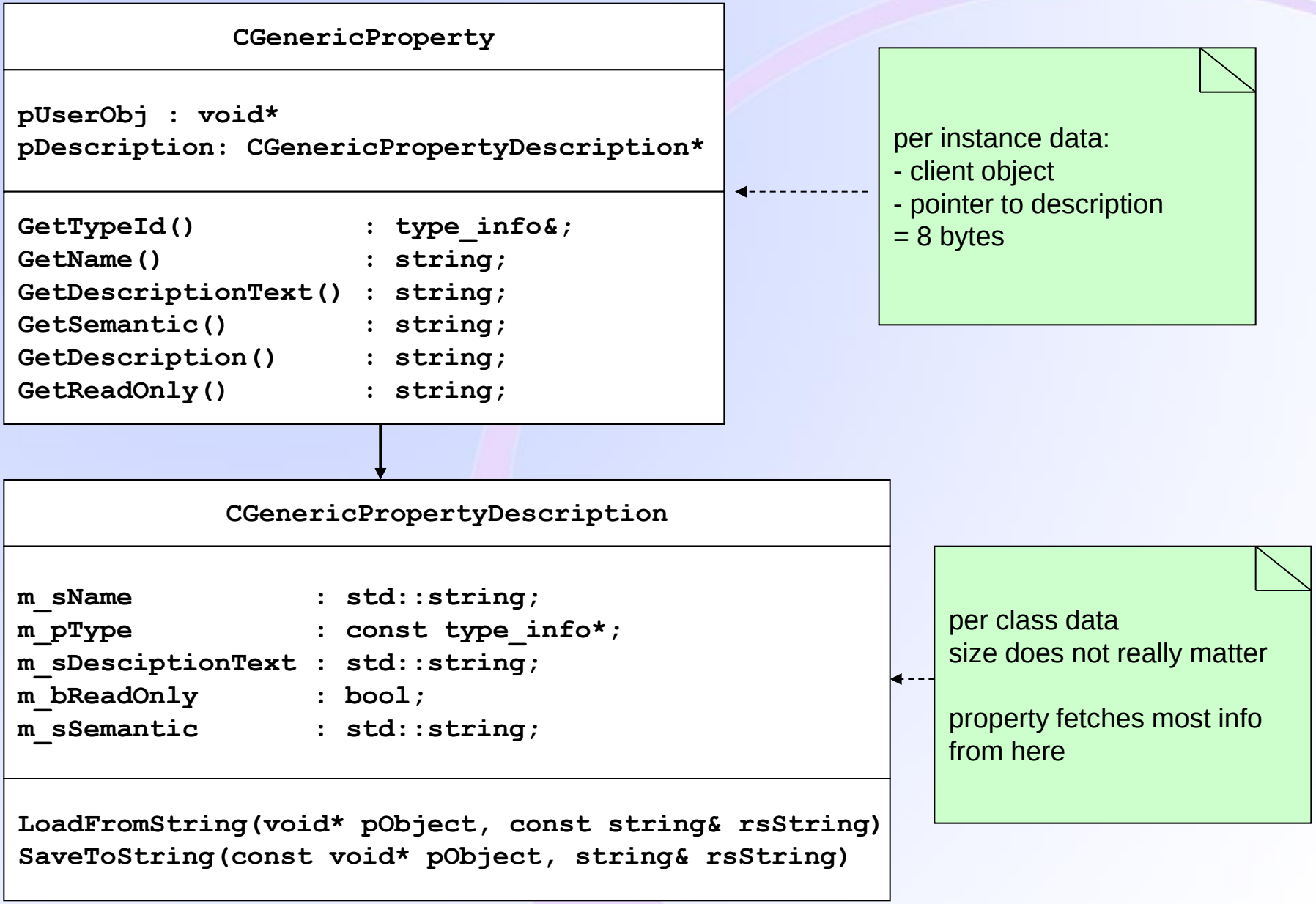
```
[LocalizableAttribute(true)]
public: virtual property DockStyle Dock
{
    DockStyle get ();
    void set (DockStyle value);
}
```

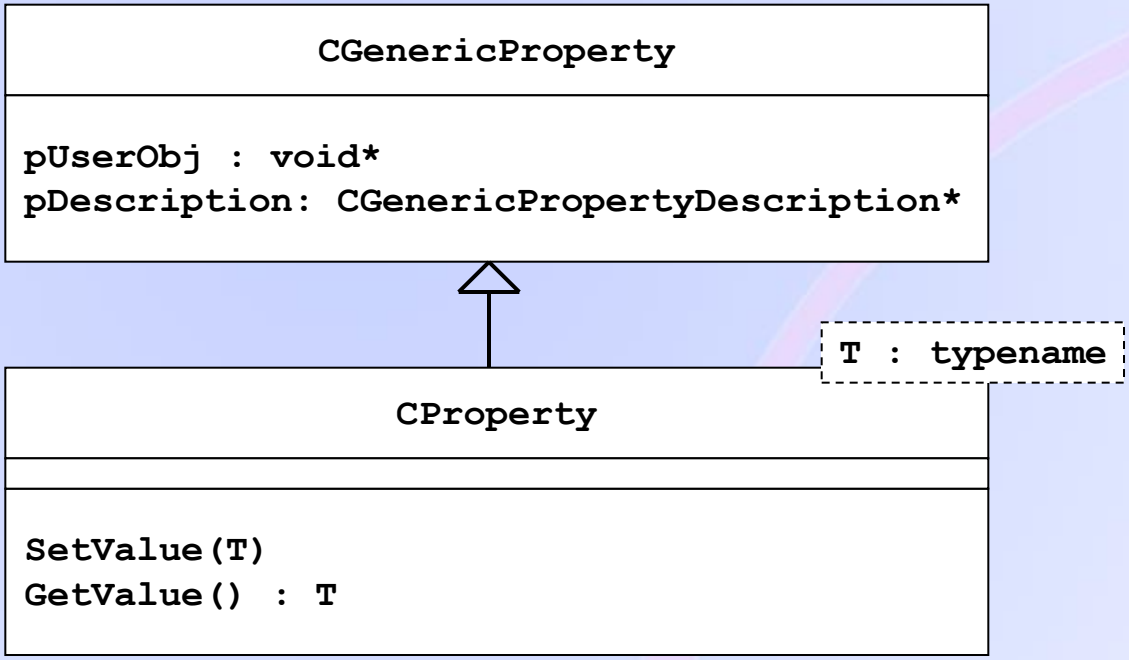


- windows forms editor clearly using **domain knowledge** to display special controls for some properties
  - can figure out the names of the enum entries using refelection, but not what „top“ or „fill“ means

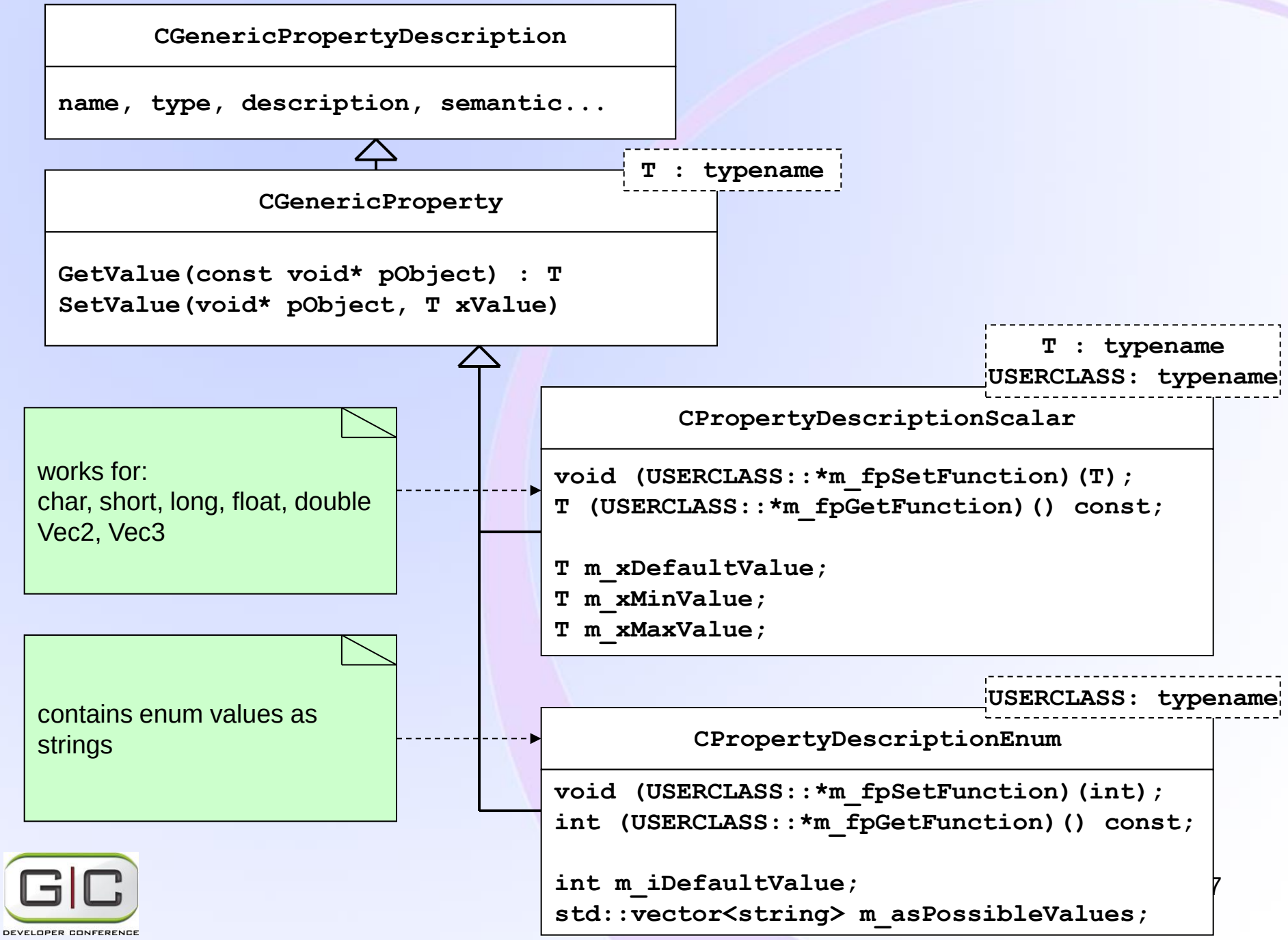
Homebrew Properties!

- accessing properties through functions → good idea!
  - clean: uses public API
  - object can react to changes
  - hides implementation from outside
- Idea 1:
  - use **member function pointers**
  - i.e. existing get/set functions
- Idea 2:
  - decouple per-instance and per-class data
  - type information needs to be kept only once per class!


CGenericProperty
pUserObj : void*
pDescription: CGenericPropertyDescription*
GetTypeId() : type_info&;
GetName() : string;
GetDescriptionText() : string;
GetSemantic() : string;
GetDescription() : string;
GetReadOnly() : string;
per instance data:
- client object
- pointer to description
= 8 bytes
CGenericPropertyDescription
m_sName : std::string;
m_pType : const type_info*;
m_sDesciptionText : std::string;
m_bReadOnly : bool;
m_sSemantic : std::string;
LoadFromString(void* pObject, const string& rsString)
SaveToString(const void* pObject, string& rsString)
per class data
size does not really matter
property fetches most info
from here



- CProperty does not add any data members!
  - can create array of CGenericProperty **values** and cast them as needed
  - not using polymorphism


CGenericPropertyDescription
name, type, description, semantic...
CGenericProperty
T : typename
GetValue(const void* pObject) : T
SetValue(void* pObject, T xValue)
works for:
char, short, long, float, double
Vec2, Vec3
T : typename
USERCLASS: typename
CPropertyDescriptionScalar
void (USERCLASS::*m_fpSetFunction)(T);
T (USERCLASS::*m_fpGetFunction)() const;
T m_xDefaultValue;
T m_xMinValue;
T m_xMaxValue;
contains enum values as
strings
USERCLASS: typename
CPropertyDescriptionEnum
void (USERCLASS::*m_fpSetFunction)(int);
int (USERCLASS::*m_fpGetFunction)() const;
int m_iDefaultValue;
std::vector<string> m_asPossibleValues;

```
class CCheckBox
{
    enum CheckBoxState { CB_Unchecked, CB_Checked, CB_Default };

    void SetChecked(int p_eChecked = CB_Checked);
    int GetChecked() const;

    void SetText(string sText);
    string GetText() const;

    void GetProperties(vector<CGenericProperty>& apxProperties) const;
};
```

```
CPropertyDescriptionString<CCheckBox, string> CCheckBox::Prop_Text(
          "Text", "Text that appears next to CheckBox", false, "",
          &CCheckBox::SetText, &CCheckBox::GetText);

CPropertyDescriptionEnum<CCheckBox> CCheckBox::Prop_State(
          "State", "Current state", false, "Unchecked", "Unchecked Checked Default",
          &CCheckBox::SetChecked, &CCheckBox::GetChecked);

void
CCheckBox::GetProperties(vector<CGenericProperty>& apxProperties) const
{
    __super::GetProperties(apxProperties);

    apxProperties.push_back(CGenericProperty(&Prop_Text, (void*) this));
    apxProperties.push_back(CGenericProperty(&Prop_State, (void*) this));
}
```

Homebrew Properties!

- semantic
  - gives editor a hint how property is used
  - a string property could be...
    - ... a file name
    - ... a font name
    - ... or just a string

- Validation / Range data
  - Different for each type
  - Scalar : min / max / default value
  - Enum: list of named values, default value
  - String: max length, allowed characters, default value

Thank you!

Questions / Comments?

if you want the slides:
david.salz@bitfield.de