Concern-based Composition and Reuse of Distributed Systems

Andrey Nechypurenko

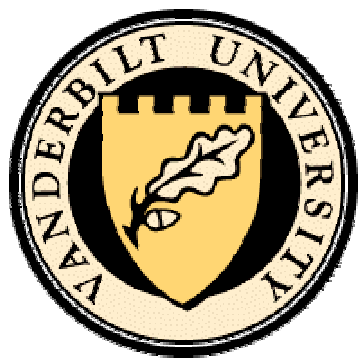
andrey.nechypurenko@siemens.com

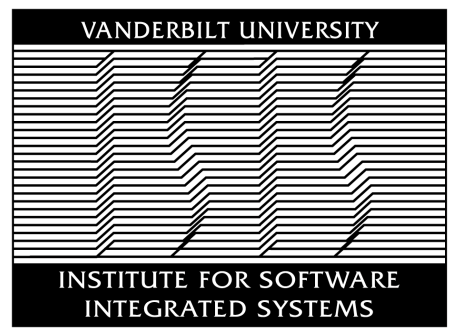
Siemens AG, Germany

Douglas Schmidt, Tao Lu, Gan Deng, Emre Turkay, Aniruddha Gokhale

Vanderbilt University
Nashville, TN 37203

S





Work supported by grant from Siemens AG, Germany

Research Synopsis

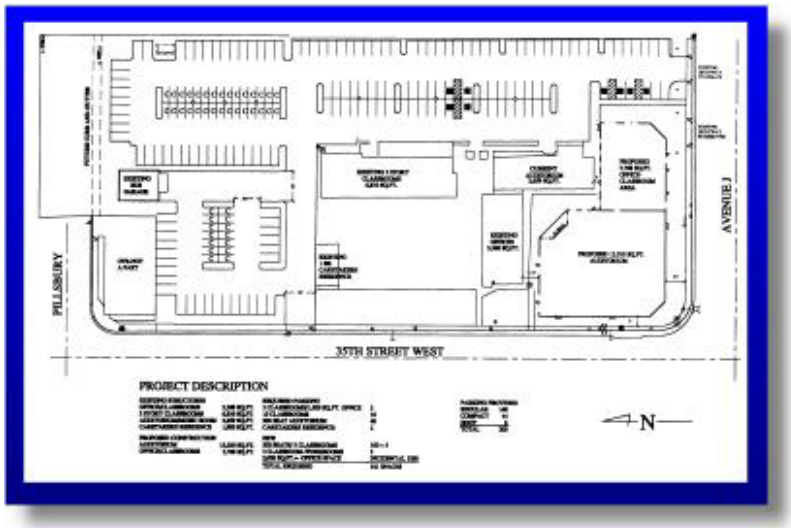
Motivation:

- Growing complexity motivates the need of higher level composition units for next generation of applications.
- Currently it is not clear how these blocks should looks like and what are the fundamental composition rules.

Goals:

- Find the typical source of complexity while composing the system out of higher level abstraction blocks.
- Define core composition rules for systems built out of the higher level abstraction blocks.

R&D Focus: Inventory Tracking System (ITS)



- **Large scale Inventory Tracking system**
- **Provides automation for**
  - **Auto Task Scheduling**
  - **Auto Good Placement Arrangement**
  - **Auto Route finding**
- **Human-Machine interaction**

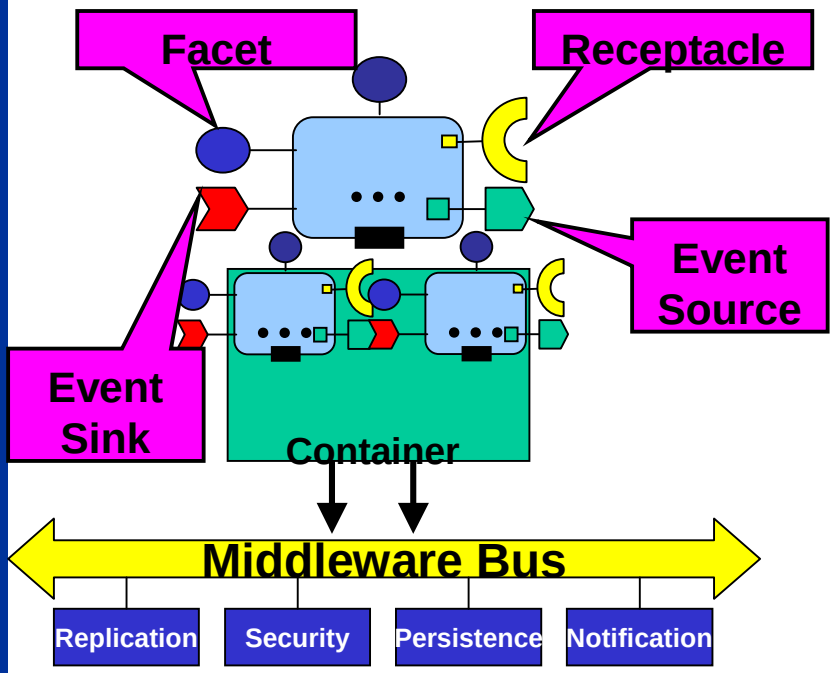
Domain Specific Requirements:

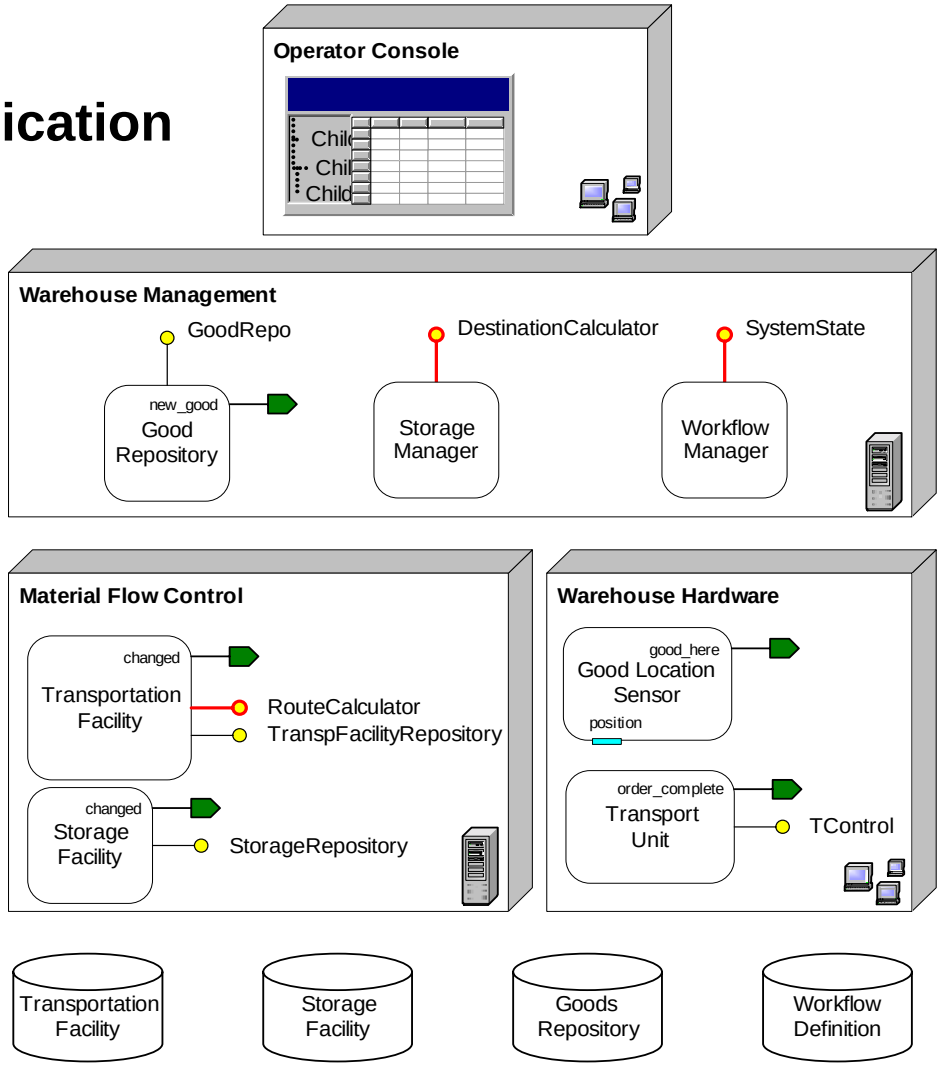
1. Flexibility
2. Reusability
3. Extensibility
4. Affordability
5. Usability

Our Solution - Component Integrated ACE ORB 1/2

Component Middleware

- Encapsulation of the application logic.
- QoS provisioning.





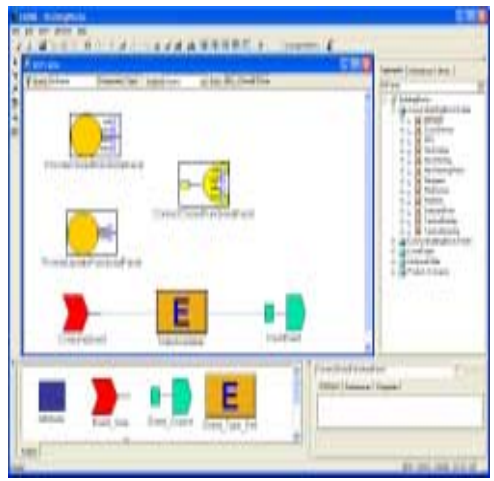
Our Solution 2/2

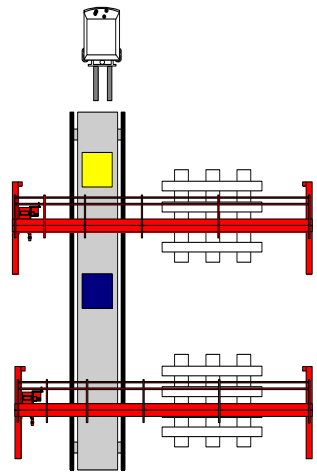
We still have the following problems.

1. Configuration of the Component Middleware
2. Configuration of warehouse equipment
3. Complexity introduced by the large scale of the warehouse system (warehouse part and component assembly)

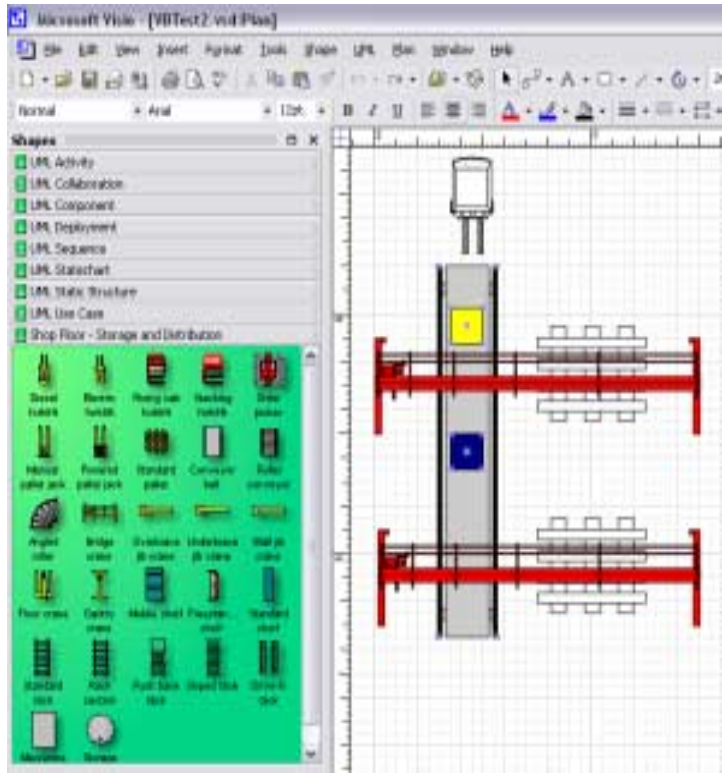
Solution strategy:

Use modeling technique to cover different configuration and initialization aspects





Domain Specific model-Warehouse Model 2/2



Warehouse configuration model defines:

- Physical location/dimension of the warehouse parts.
- Customized properties of the parts. (Name, type etc.)

The model interpreter dose:

- Enumerating all relative objects and populate the database.
- Construct the connection graph according to the location, dimension and type information in the model and store them in the relational table.

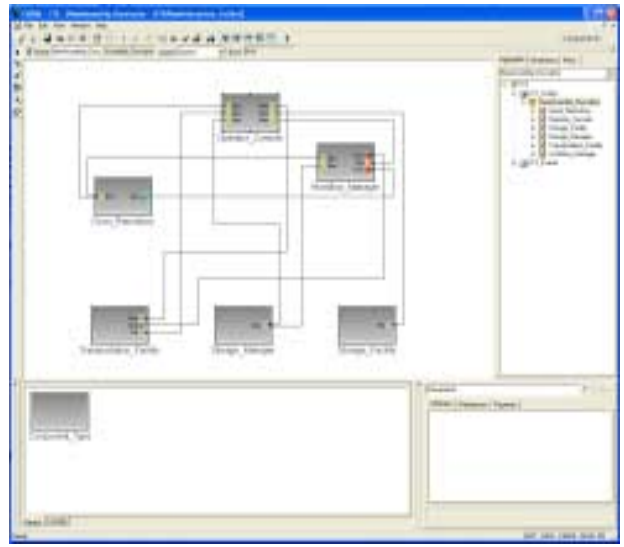
Component Model with GME

The Component model defines:

- Application component related information. (Interface, location etc.)
- Inter-connection between components.
- QoS configuration. (on-going work)

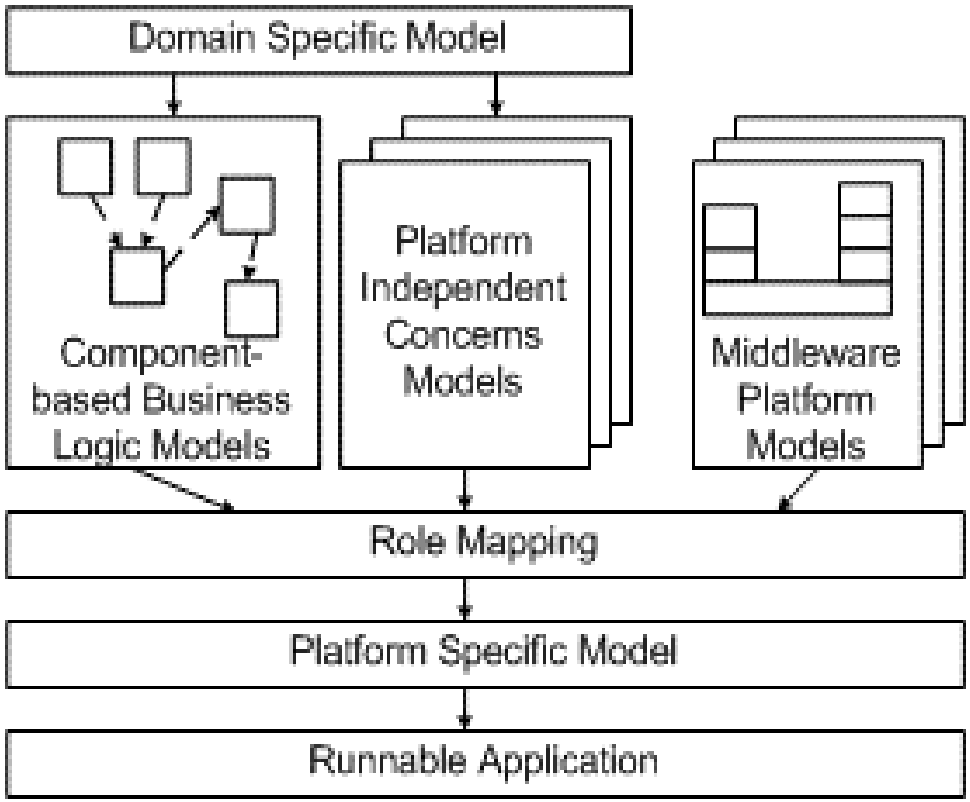
The model interpreter does:

- Generate schema files to do deploy & assemble components.
- Check the consistency for connections.
- Generate middleware configuration code. (on-going work)

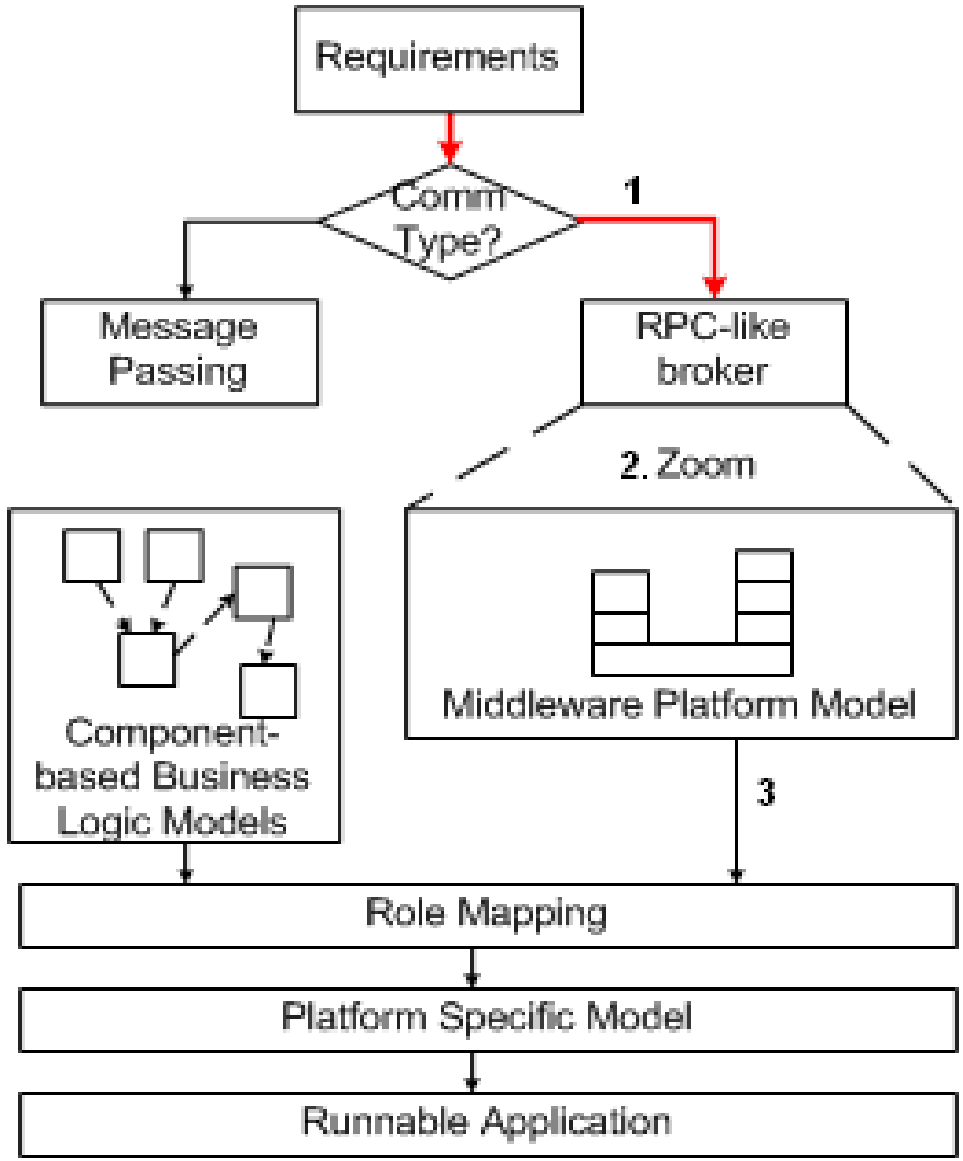


Integrated Concern Modeling and Manipulation Environment (ICMME)

- **Pattern Based Role Mapping**
- **Combine AOSD and OMG's EDOC standard**



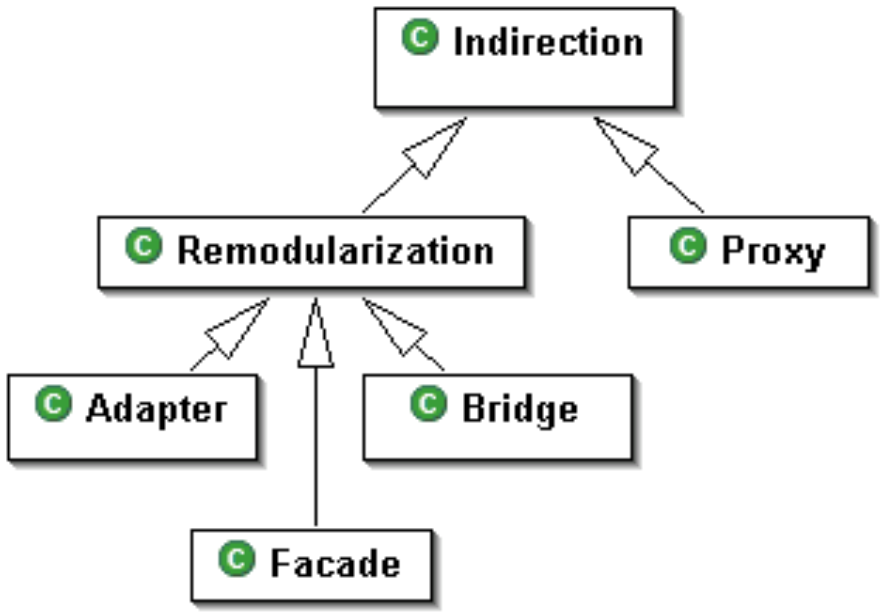
ICMME Example – Broker Pattern

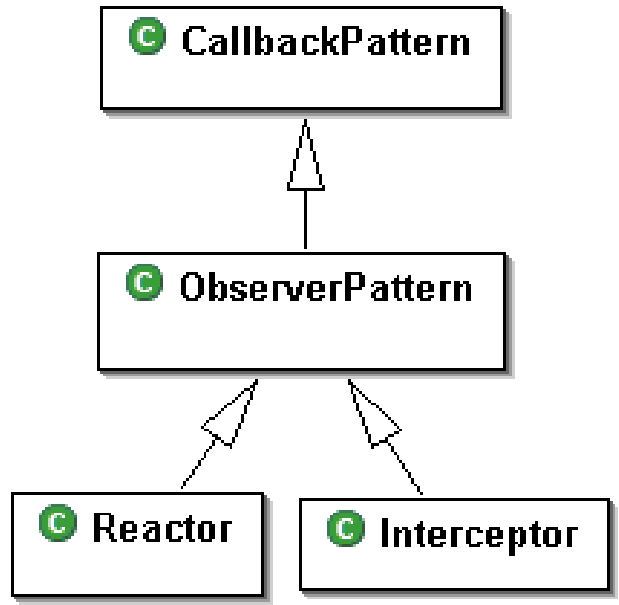


Foundation for Pattern Based Composition

If we want to easily manipulate with higher level blocks, the similar set of relationships as in OO is required.

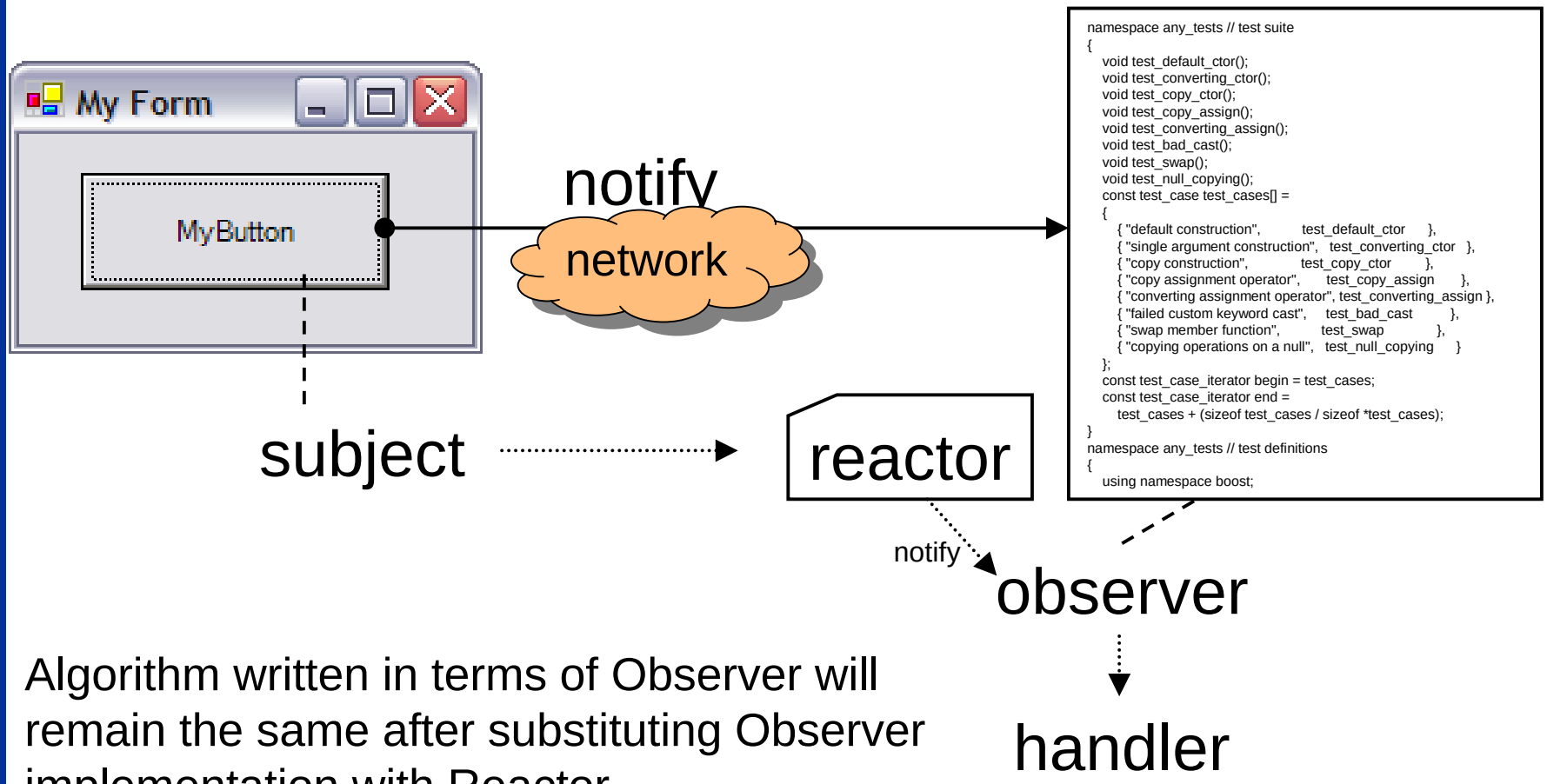
Patterns have some higher level relationship.
This relationship could be called inheritance.





Concern Manipulation Example

Observer relationships



Algorithm written in terms of Observer will remain the same after substituting Observer implementation with Reactor.

It is possible because of inheritance relationships between these two patterns and is in accordance with LSP.

Gap between problem and solution domain

too big

```
namespace any_tests // test suite
{
   void test_default_ctor();
   void test_converting_ctor();
   void test_copy_ctor();
   void test_copy_assign();
   void test_converting_assign();
   void test_bad_cast();
   void test_swap();
   void test_null_copying();
   const test_case test_cases[] =
   {
      { "default construction",          test_default_ctor     },
      { "single argument construction",  test_converting_ctor  },
using namespace boost;
```

Problem Domain exposed with DSL

Solution Domain

Mapping from the problem domain to solution domain need to overcome a lot of details which are typically addressed in code generator and is a primary source of complexity.

Gap between problem and solution domain

Problem Domain
exposed with DSL

DSL1

DSL2

```
namespace any_tests // test suite
{
   void test_default_ctor();
   void test_converting_ctor();
   void test_copy_ctor();
   void test_copy_assign();
   void test_converting_assign();
   void test_bad_cast();
   void test_swap();
   void test_null_copying();
   const test_case test_cases[] =
   {
      { "default construction",          test_default_ctor     },
      { "single argument construction",  test_converting_ctor  },
using namespace boost;
```

Solution Domain

To solve the problem of growing complexity at code generator level and improve the reusability of models and generators we suggest intermediate levels of models which refines the model at higher level and is exposed with corresponding DSL.

Conclusions

- Patterns or role-based description of certain solution could be used as a higher level building blocks.
- There are higher level relationships between patterns and they could be useful to support variability and improve reusability
- In order to reduce complexity at generator level and improve model and generator reusability, the mapping from problem domain to solution domain should be split with intermediate models exposed with the set of DSLs.

Lessons Learned

1. No single tool is enough to cover all modeling needs – set of standard-compliant interoperable tools is necessary (MOF).
2. UML is not always the best way to represent the Domain Specific Modeling Language
3. Do not try to substitute programming language with boxes and bubbles – one of the ultimate goals and benefits of the models is to raise the abstraction level
4. Pay attention to the model interpreters development – they can grow even faster and become more complex then the system you develop