IBM Software Group | Rational. software

# Model-Driven Development: Its Essence and Opportunities

Bran Selic
IBM Distinguished Engineer
IBM Rational Software – Canada
bselic@ca.ibm.com

# A Bit of Modern Software...

```
SC_MODULE(producer)
{
sc_outmaster<int> out1;
sc_in<bool> start; // kick-start
void generate_data ()
{
for(int i =0; i <10; i++) {
out1 =i ; //to invoke slave;}
}
SC_CTOR(producer)
{
SC_METHOD(generate_data);
sensitive << start;}};
SC_MODULE(consumer)
{
sc_inslave<int> in1;
int sum; // state variable
void accumulate (){
sum += in1;
cout << "Sum = " << sum << endl;}
SC_CTOR(consumer)
{
SC_SLAVE(accumulate, in1);
sum = 0; // initialize
};
SC_MODULE(top) // container
{
producer *A1;
consumer *B1;
sc_link_mp<int> link1;
SC_CTOR(top)
{
A1 = new producer("A1");
A1.out1(link1);
B1 = new consumer("B1");
B1.in1(link1);}};
```

Can you see the architecture?

# ...and its Model


Can you see it now?

# Back to Code...

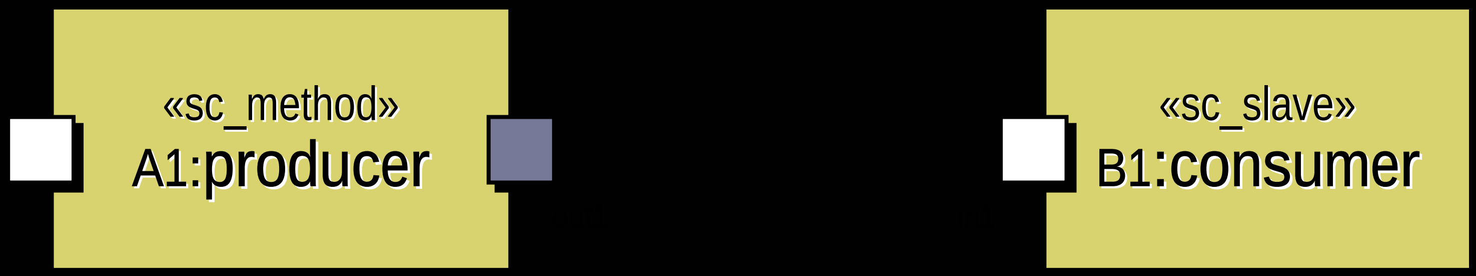
```
SC_MODULE(producer)
{
sc_outmaster<int> out1;
sc_in<bool> start; // kick-start
void generate_data ()
{
for(int i =0; i <10; i++) {
out1 =i ; //to invoke slave;}
}
SC_CTOR(producer)
{
SC_METHOD(generate_data);
sensitive << start;}};
SC_MODULE(consumer)
{
sc_inslave<int> in1;
int sum; // state variable
void accumulate (){
sum += in1;
cout << "Sum = " << sum << endl;}
SC_CTOR(consumer)
{
SC_SLAVE(accumulate, in1);
sum = 0; // initialize
};
SC_MODULE(top) // container
{
producer *A1;
consumer *B1;
sc_link_mp<int> link1;
SC_CTOR(top)
{
A1 = new producer("A1");
A1.out1(link1);
B1 = new consumer("B1");
B1.in1(link1);}};
```

# Breaking the Architecture….

```
SC_MODULE(producer)
{
sc_outmaster<int> out1;
sc_in<bool> start; // kick-start
void generate_data ()
{
for(int i =0; i <10; i++) {
out1 =i ; //to invoke slave;}
}
SC_CTOR(producer)
{
SC_METHOD(generate_data);
sensitive << start;}};
SC_MODULE(consumer)
{
sc_inslave<int> in1;
int sum; // state variable
void accumulate (){
sum += in1;
cout << “Sum = “ << sum << endl;}
SC_CTOR(consumer)
{
SC_SLAVE(accumulate, in1);
sum = 0; // initialize
};
SC_MODULE(top) // container
{
producer *A1;
consumer *B1;
sc_link_mp<int> link1;
SC_CTOR(top)
{
A1 = new producer(“A1”);
//A1.out1(link1);
B1 = new consumer(“B1”);
//B1.in1(link1);}};
```

Can you see where?

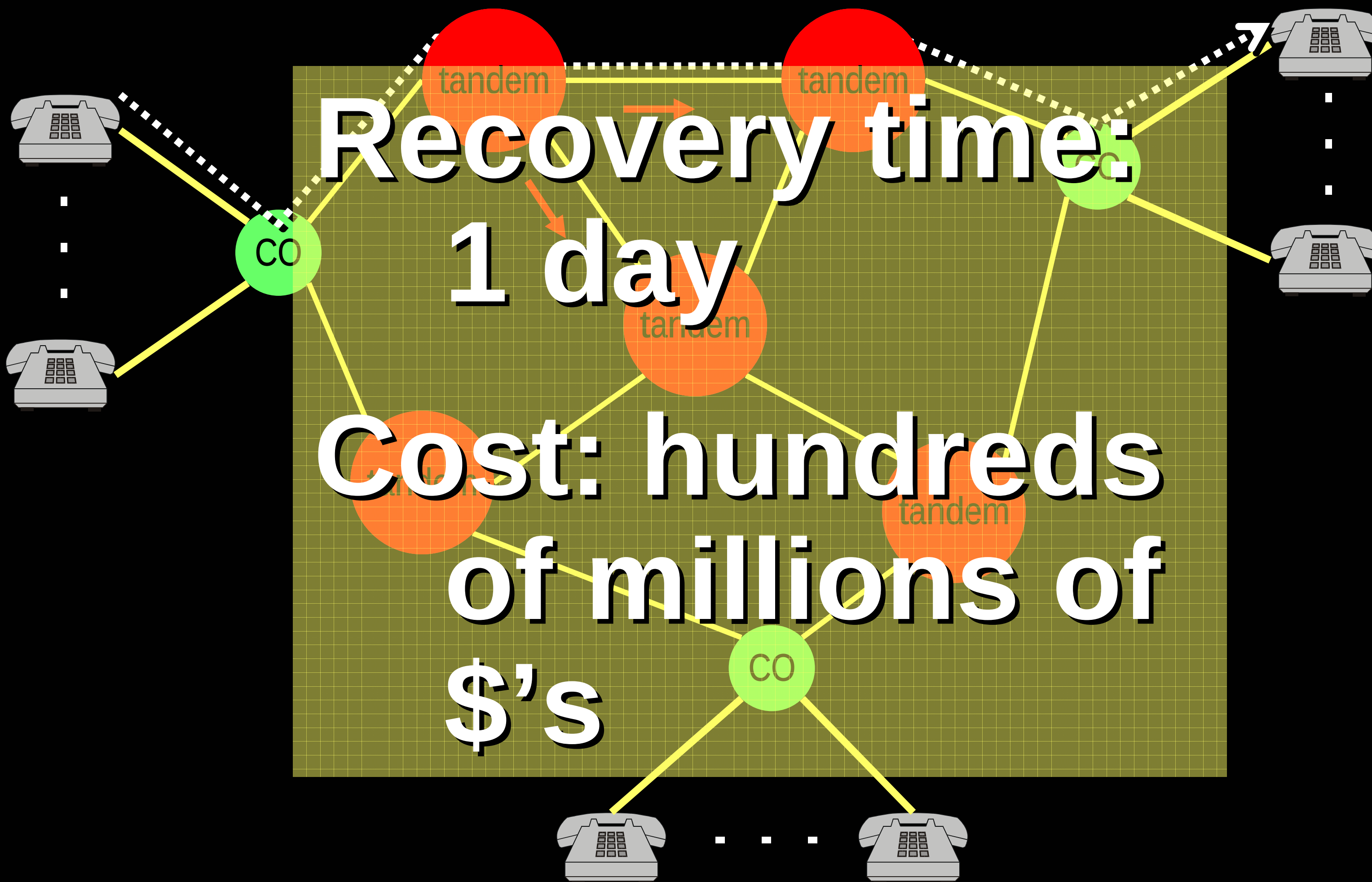
# Breaking the Architecture....




Can you see it now?

# A Major Engineering Disaster

- 1990: AT&T Long Distance Network (Northeastern US)




**Recovery time: 1 day**

**Cost: hundreds of millions of $'s**

# The Culprit

- Missing “break” statement in a software module
  - One (1) missing line among millions (X,000,000)

```
. . .;
switch (...) {
      case a : ...;
            break;
      case b :...;
            break;
            . . .
      case m : ...;
      case n : ...;
            . . .
};
```

**Execution fell through unintentionally into the next case**

# Q: Why is Writing Correct Software so Difficult?

# A: COMPLEXITY!

*Modern software is reaching levels of complexity encountered in biological systems; sometimes comprising systems of systems each of which may include tens of millions of lines of code*

…any one of which may bring down the entire system at great expense

# Fred Brooks on Complexity

- [From: F. Brooks, "*The Mythical Man-Month*", Addison Wesley, 1995]
- *Essential complexity*
  - inherent to the problem
  - cannot be eliminated by technology or technique
  - e.g., the algorithmic complexity of the "traveling salesman" problem
- *Accidental complexity*
  - due to technology or methods used to solve the problem
  - e.g., building a skyscraper using only hand tools

# The Issue

- *Most mainstream programming languages abound in accidental complexity*
  - Including “modern” OO languages (Java, C#, …)
- *Programs require significant intellectual effort to understand*
- *Defect intolerant:* with a *chaotic* quality:
  - The effects of *barely perceptible* flaws cannot be predicted
  - …but, they can be catastrophic

# The Impact

- Abstraction of software is both difficult and risky
  - Eliminates our most effective means for managing complexity
- Our ability to exploit formal mathematical methods is greatly diminished
  - Foundation of all modern engineering disciplines
  - Reason: Mathematical methods depend on abstraction to avoid practical computability hurdles

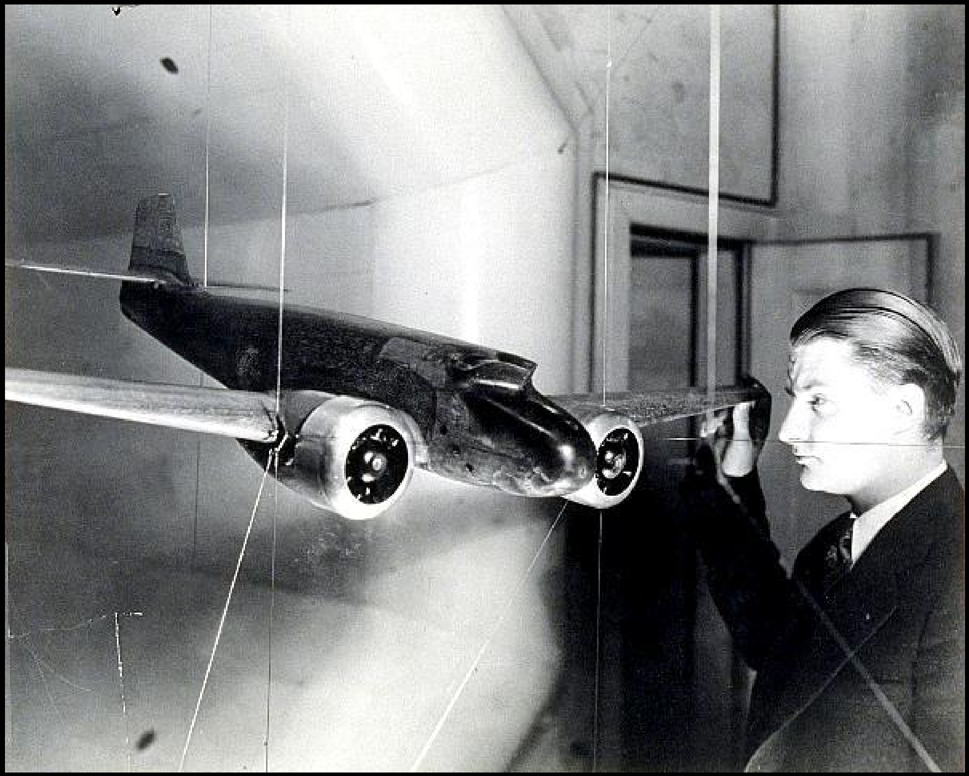
# The Model and the Code

```
SC_MODULE(producer)
{
sc_outmaster<int> out1;
sc_in<bool> start; // kick-start
void generate_data ()
{
for(int i =0; i <10; i++) {
out1 =i ; //to invoke slave;}
}
SC_CTOR(producer)
{
SC_METHOD(generate_data);
sensitive << start;}};
SC_MODULE(consumer)
{
sc_inslave<int> in1;
int sum; // state variable
void accumulate (){
sum += in1;
cout << “Sum = “ << sum << endl;}
SC_CTOR(consumer)
{
SC_SLAVE(accumulate, in1);
sum = 0; // initialize
};
SC_MODULE(top) // container
{
producer *A1;
consumer *B1;
sc_link_mp<int> link1;
SC_CTOR(top)
{
A1 = new producer(“A1”);
A1.out1(link1);
B1 = new consumer(“B1”);
B1.in1(link1);}};
```

# Use of Models in Engineering

- Probably as old as engineering (c.f., Vitruvius)
- Engineering model:

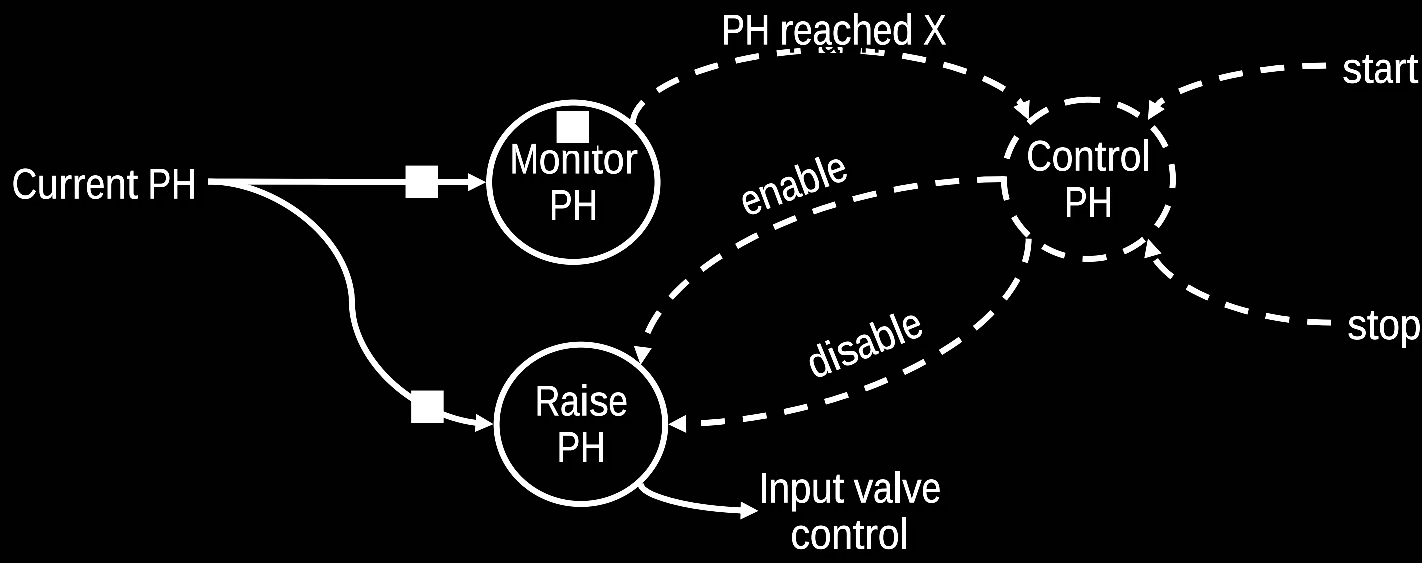
  *A reduced representation of some system that highlights its properties of interest from a given viewpoint*



- We don't see everything at once
- What we do see is adjusted to human understanding

**What about modeling software?**

# A Common View of Software Modeling




*"…bubbles and arrows, as opposed to programs, …never crash"*

-- B. Meyer
*"UML: The Positive Spin"*
American Programmer, 1997

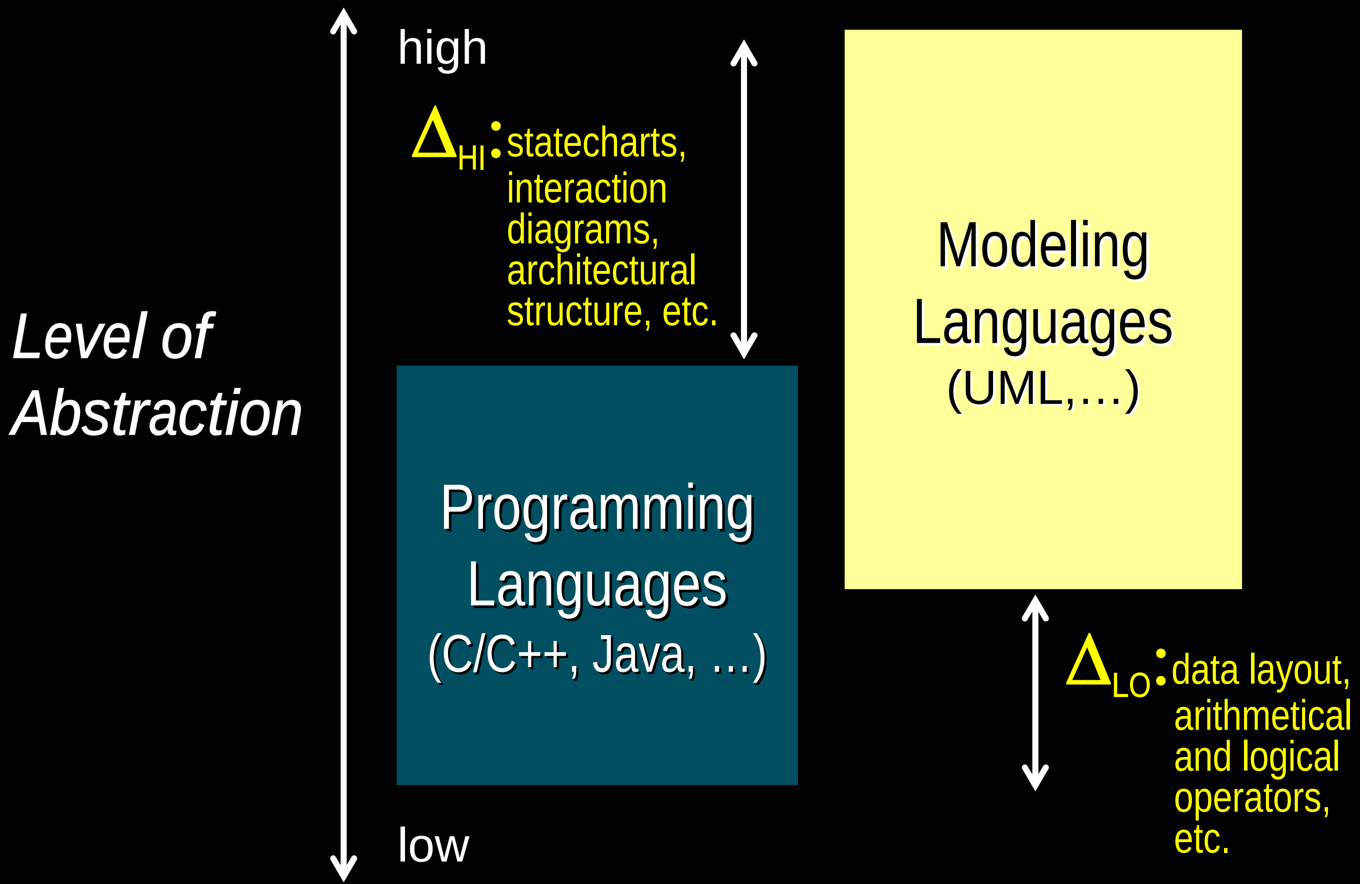
# Key Characteristics of Useful Engineering Models

1. Abstract
   - Emphasize important aspects while obscuring irrelevant ones
2. Understandable
   - Expressed in a form that is readily understood by observers
3. Accurate
   - Faithfully represents the modeled system
4. Predictive
   - Can be used to answer questions about the modeled system
5. Cost effective
   - Much be cheaper to construct and study than the modeled system

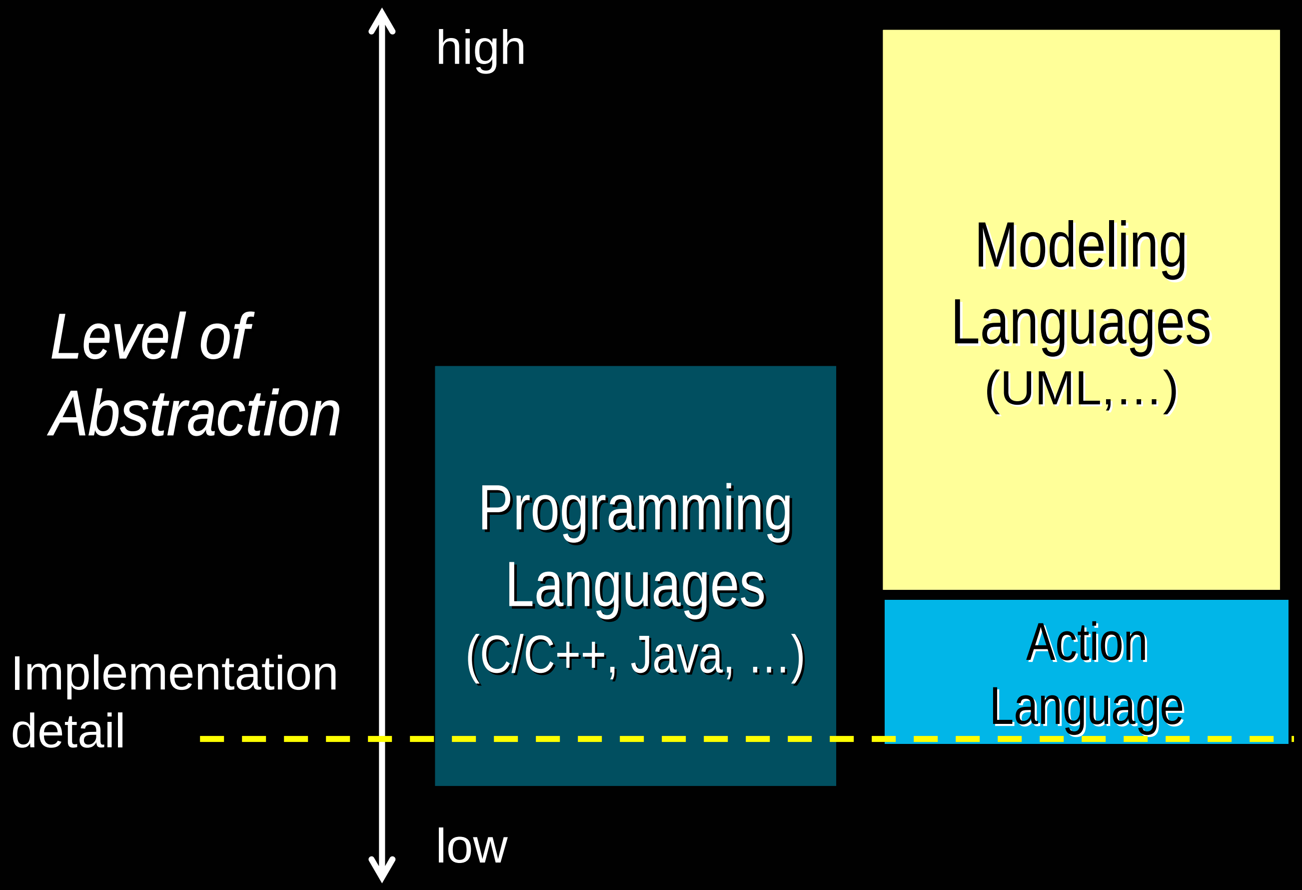
# The Key Issue for Software Modeling

*How can we make our software models more accurate?*

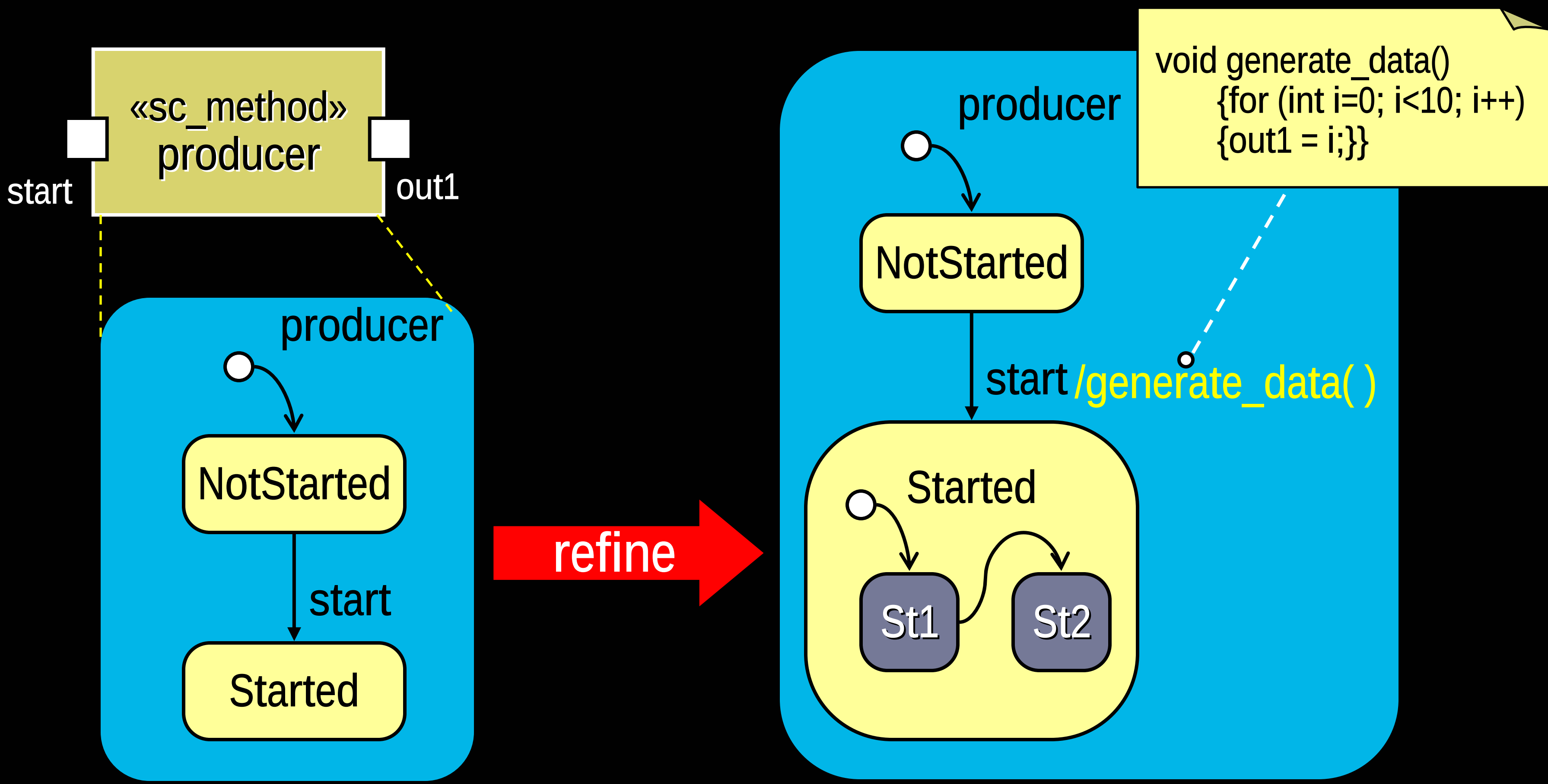
# Modeling Languages vs Programming Languages



# Software Models: Filling in the Detail



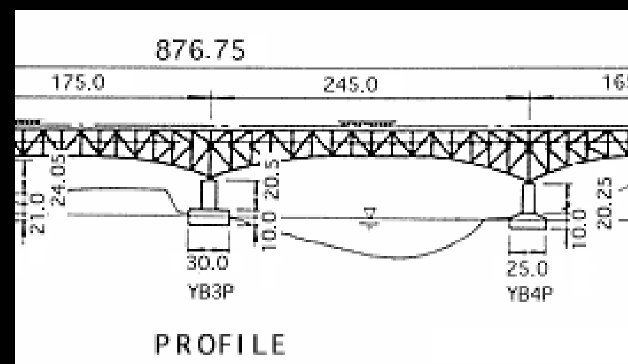
# Model Evolution: Refinement



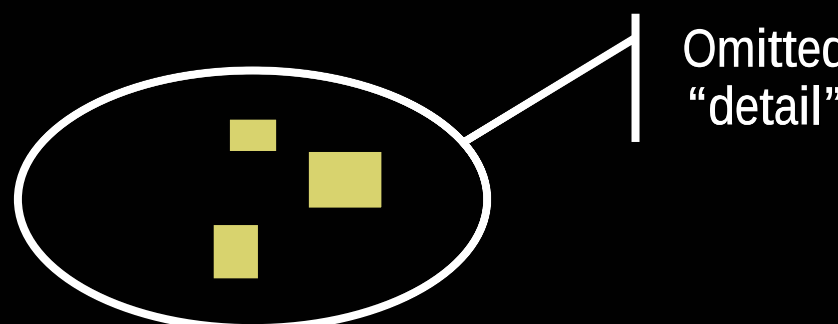

- Models can be refined continuously until the application is fully specified ⇒ *the model becomes the system that it was modeling!*

# Models and Reality

- The abstraction quality of models that is key to their greatest benefit is also the source of their greatest vulnerability

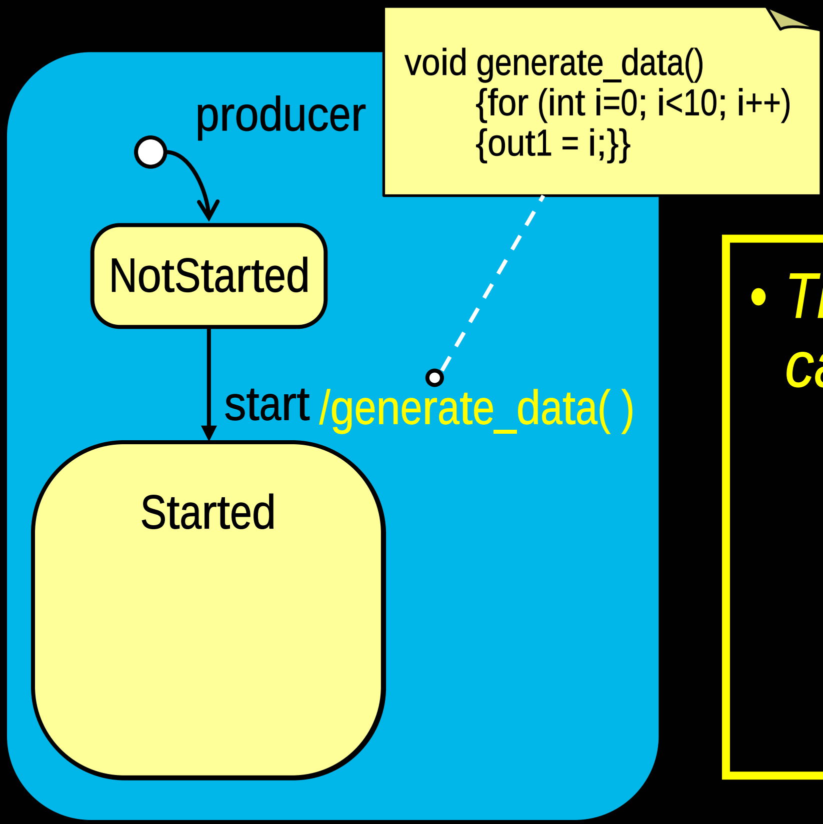



- The model and the modeled system are different entities $\Rightarrow$ loss of accuracy



# Models of Software

- *Uniquely, in software, an abstraction can be extracted <u>automatically</u> <u>from the system itself</u> through suitable transformations*




- *The computer offers a uniquely capable abstraction device:*

  *Software can be represented from any desired viewpoint at any desired level of abstraction*

  *<u>The abstraction is in the system</u>*

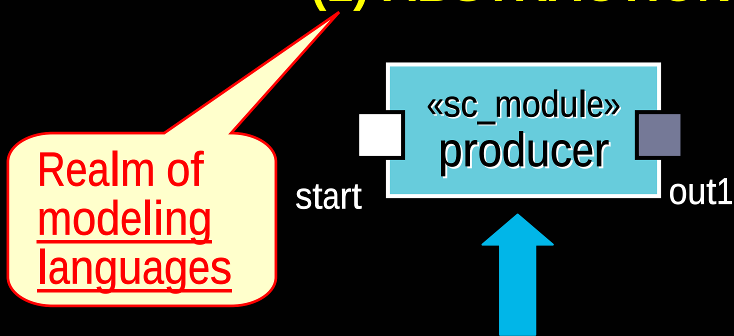
# The Remarkable Thing About Software

*Software has the rare property that it allows us to directly evolve models into complete implementations <u>without discontinuities in the expertise, materials, tools, or methods</u>!*

# Model-Driven Development (MDD)

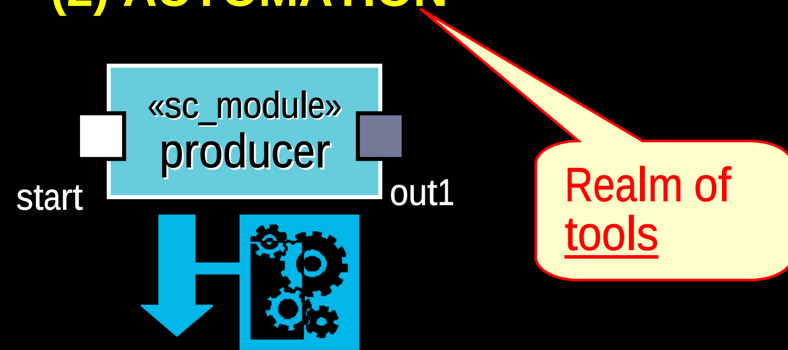
- An approach to software development in which the focus and primary artifacts of development are models (vs programs)
- Based on two time-proven methods:

**(1) ABSTRACTION**




```
SC_MODULE(producer)
{sc_inslave<int> in1;
int sum; //
void accumulate (){
sum += in1;
cout << "Sum = " <<
sum << endl;}
```

**(2) AUTOMATION**




```
SC_MODULE(producer)
{sc_inslave<int> in1;
int sum; //
void accumulate (){
sum += in1;
cout << "Sum = " <<
sum << endl;}
```

# Model-Driven Architecture (MDA)

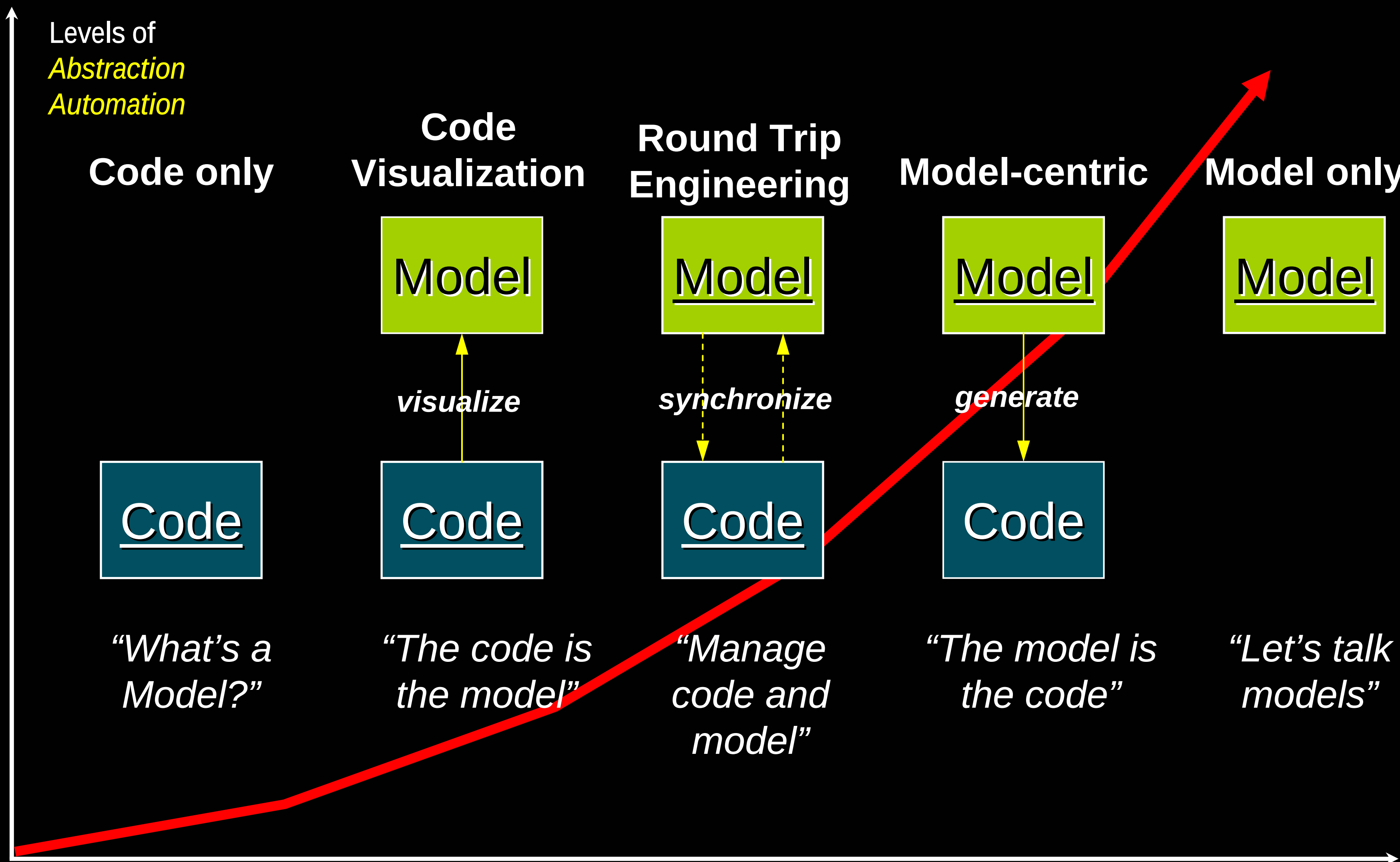
- An OMG initiative to support model-driven development through a series of open standards

**(1) ABSTRACTION** **(2) AUTOMATION**


**(3) OPEN STANDARDS**

- ***Modeling languages***
- ***Interchange standards***
- ***Model transformations***
- ***Software processes***
- ***etc.***

# The Specturm of MDD

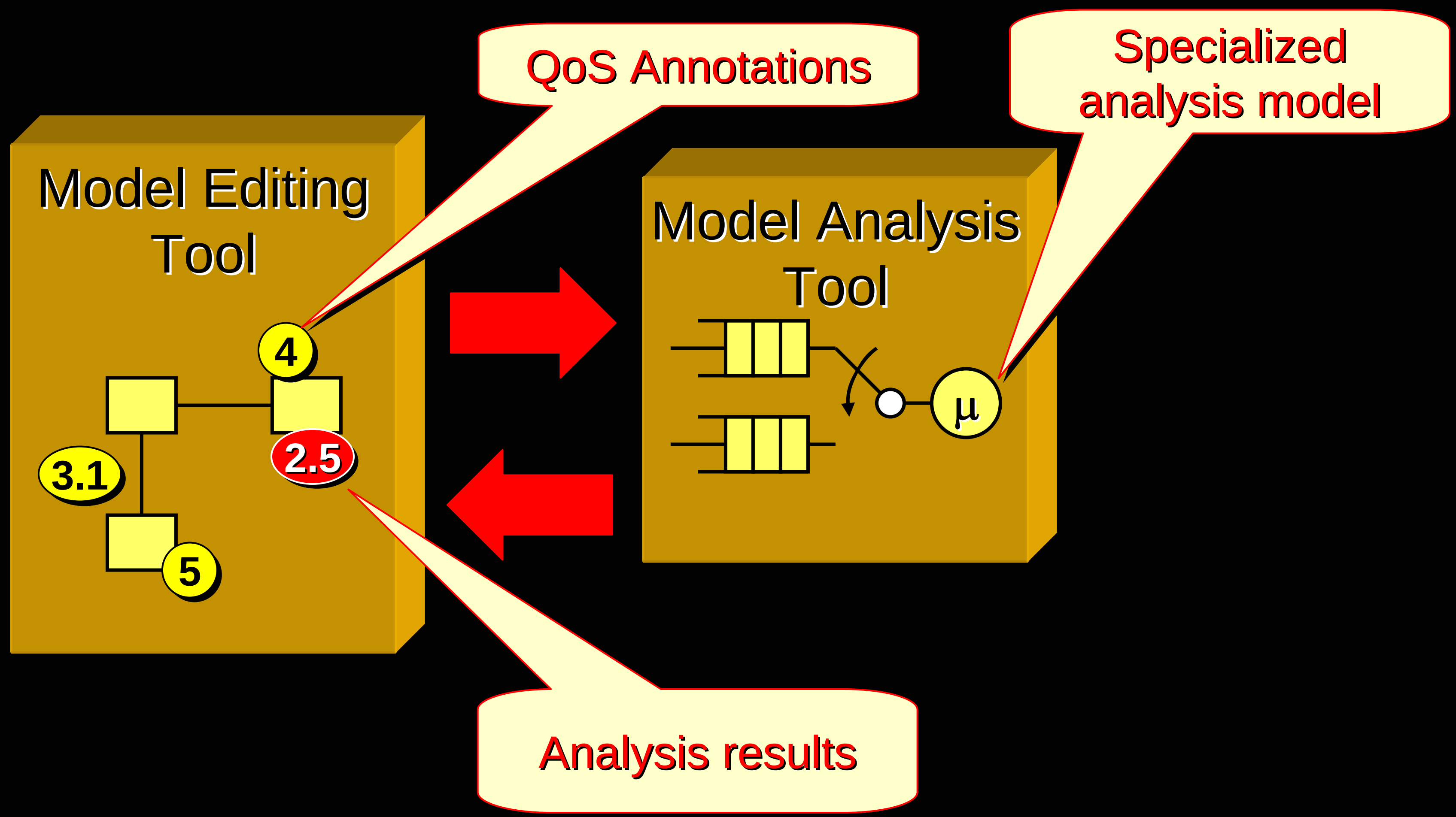


# Opportunity: Automated Formal Validation

- Checking for safety and liveness properties of a software design
  - Safety: Good things WILL happen
  - Liveness: Bad things will NOT happen
- Modeling language constructs can be chosen to avoid the semantic complexity of programming languages
  - Basing them on well-understood and well-behaved formalisms such as state machines and Petri nets
  - Enables formal model checking, theorem proving
- Badly in need of a theory of modeling language design

# Opportunity: Automated Model Analysis

- Assessing the quantitative aspects of a software design



# Opportunity: Automatic Code Generation

```
SC_MODULE(producer)
{
sc_outmaster<int> out1;
sc_in<bool> start; // kick-start
void generate_data ()
{
for(int i =0; i <10; i++) {
out1 =i ; //to invoke slave;}
}
SC_CTOR(producer)
{
SC_METHOD(generate_data);
sensitive << start;}};
SC_MODULE(consumer)
{
sc_inslave<int> in1;
int sum; // state variable
void accumulate (){
sum += in1;
cout << "Sum = " << sum << endl;}
```

```
SC_CTOR(consumer)
{
SC_SLAVE(accumulate, in1);
sum = 0; // initialize
};
SC_MODULE(top) // container
{
producer *A1;
consumer *B1;
sc_link_mp<int> link1;
SC_CTOR(top)
{
A1 = new producer("A1");
A1.out1(link1);
B1 = new consumer("B1");
B1.in1(link1);}};
```

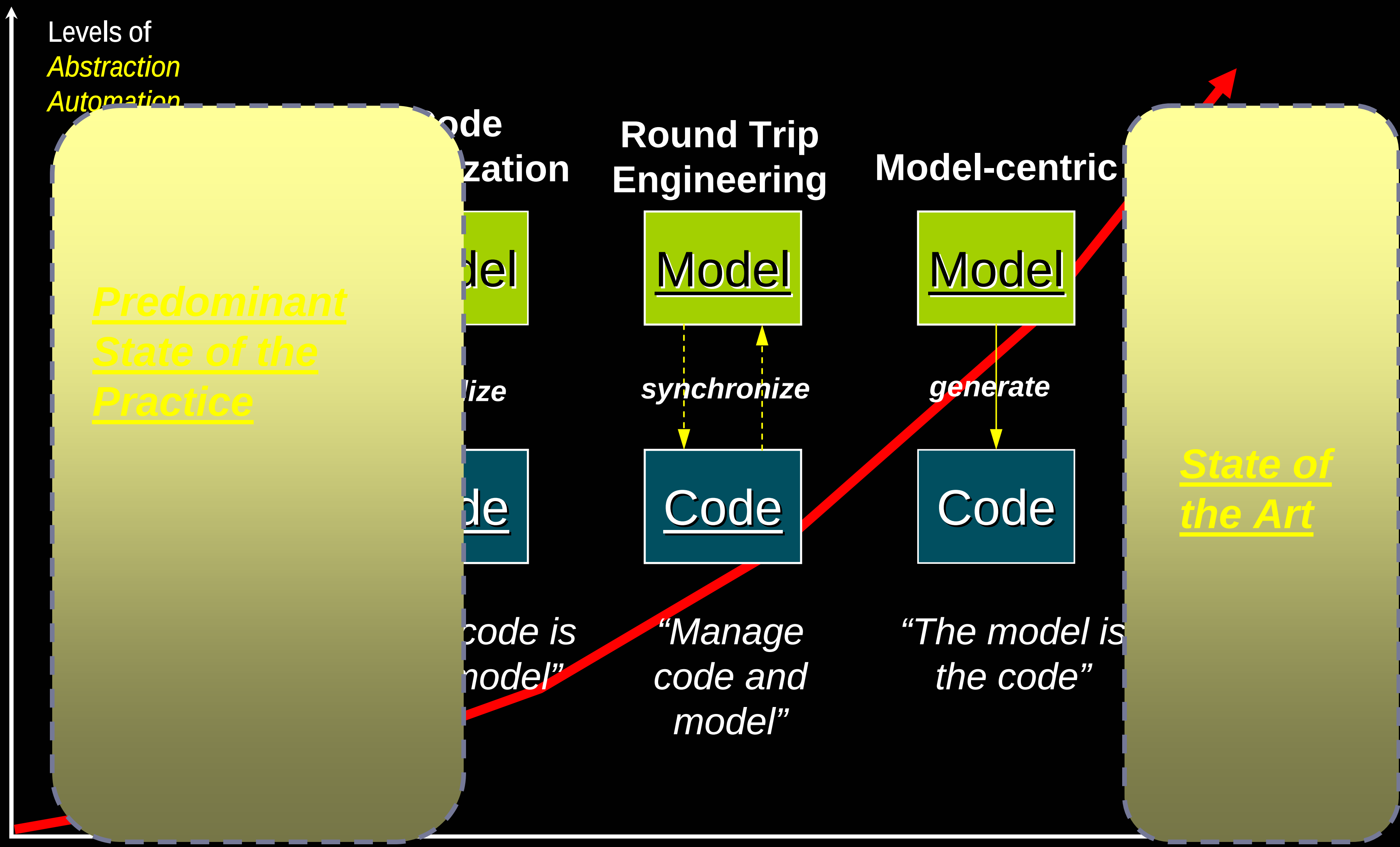
# Automatic Code Generation

- A kind of model transformation
- State of the art:
  - All development done via the model (i.e., no modifications of generated code)
  - Size: Systems equivalent to ~ 10 MLoC
  - Scalability: teams involving hundreds of developers
  - Performance: within ±5-15% of equivalent manually coded system
- Badly in need of model transform theory

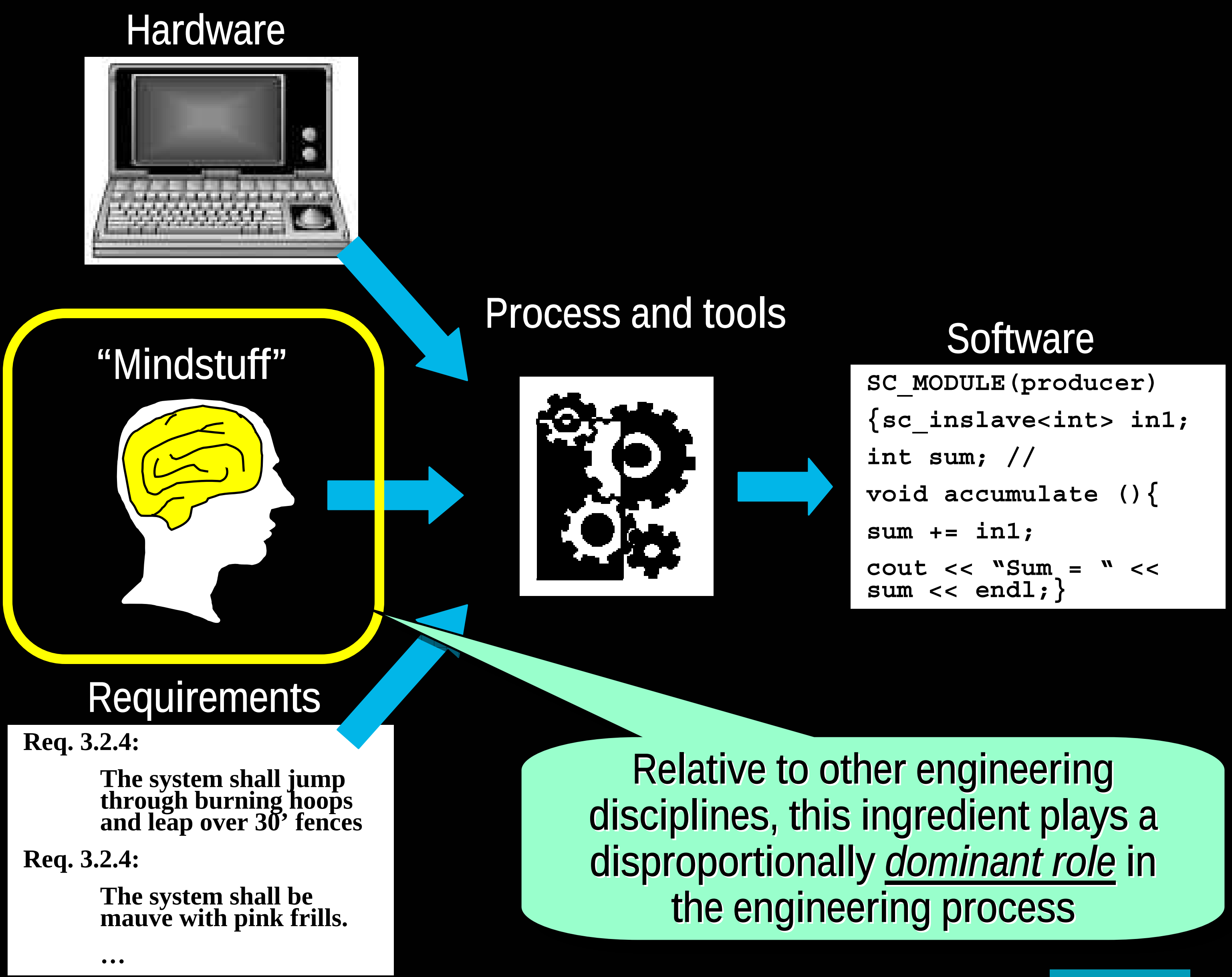
# Opportunity: Executable Models

- D. Harel: “Models that are not executable are like cars with no engines”
  - Rapid evaluation of ideas
  - Developing direct experience and insight with a problem domain
  - Boosts confidence and reduces risk
- Key capabilities
  - Controllability: ability to start/stop/slow down/speed up/drive execution
  - Observability: ability to view execution and state in model (source) form
  - Partial model execution: ability to execute abstract and incomplete models
- Opportunity: executable standard specifications

# The State of the Art and the State of the Practice



# The Idiosyncratic Nature of Software



# Some Consequences

- Products are much less hampered by physical reality
  - …but, not completely free
- The effects of aptitude differences between individuals are strongly accentuated
  - Productivity of individuals can differ by an order of magnitude
  - Not necessarily a measure of quality
  - …or intelligence
- The path from conception to realization is exceptionally fast and easy (edit-compile-run cycle)
  - Often leads to an impatient state of mind
  - …which leads to unsystematic and hastily conceived solutions (hacking)
  - Also yields a highly seductive and engrossing experience
  - …so that, often, *the medium (programming) becomes the message (reason for programming)*

# What is Engineering?

Engineering *(Merriam-Webster Collegiate Dictionary)* :

> *the application of science and mathematics by which the properties of matter and the sources of energy in nature are made useful to people*

# Why "Software" Engineering?

- Misleading term
  - The objective is not to develop software but useful systems
  - Software should be just one of the tools used by engineers for solving engineering problems
- Consequences:
  - Software engineers often identify themselves not by their problem- domain expertise (e.g., telecom, financial systems, aerospace) but by their technology expertise (e.g., C++, EJB, Linux)

    *"To a hammer all problems look like nails"*
  - Technology obsolescence and suboptimal solutions
  - *Exceptional level of resistance to technological paradigm shifts*

# Need: Getting Closer to the End User

- There is an unfortunate lack of awareness of and respect for end users
  - Personal gratification should not come solely from having designed and constructed the system, but from seeing it in use
- Implies achieving a deep level of understanding of the value of the system to the customer
  - Implies a scope of skills and knowledge that extends far beyond the technical domain
  - Required at every level (not just system architects)

# Need: Understanding the Business Case

- There is often a justifiable reason why the “best” technical solution is not the best solution for a given situation
  - E.g., cost of retraining
  - Perhaps the most frequent (and most futile) complaint of software developers worldwide
  - Based on the assumption that technical concerns (e.g., technical elegance) are always paramount
  - Often reflects a lack of awareness of overriding non-technical issues
- Software engineers and developers must be trained to understand and appreciate the greater business context

# Need: Abstraction Skills

- Abstraction plays a central role in software
  - More so than any other engineering discipline
- Mathematics is an excellent foundation for developing and honing abstraction skills
  - …and may even be directly applicable to the technical problems at hand ☺
    - Mathematical logic
    - Probability theory
    - Discrete mathematics
    - Optimization theory
    - Formal proof methods

# Summary

- Software is a unique engineering medium because of its capacity to convert abstractions into reality (and back)
- This potential is significantly hampered by the “infantile disorders” of mainstream programming technologies and mindsets
- MDD provides a unique opportunity to overcome many of these accidental complexity issues
- However, if MDD is to be successful, we must develop a proper theory of modeling and, perhaps more importantly, we must overcome the culture gap that stands in its way

# A Sampling of MDD Research Challenges

IBM

- **Theory of modeling language design**
  - Specifying the abstract syntax of modeling languages (metamodeling?)
  - Specifying the semantics of modeling languages
  - Notation (concrete syntax)
  - Specialization of modeling languages (profiles, etc.)
  - Domain-specific languages
  - View definition and reconciliation
  - Hybrid and co-design languages
  - Dynamic system modeling
  - Platform modeling and mapping
- **Theory of model transformations**
  - Horizontal, vertical, inverse
  - View extraction
  - Optimization techniques for code generation
  - etc.
- **Model analysis**
  - Qualitative analyses (safety/liveness)
  - Quantitative analyses (performance, availability, timeliness, security, etc.)
  - Testing
- **Model synthesis**
- **Automation support (tools)**
  - Usability
  - etc.