



Universities and Industry

- Existing problems with teaching Scientific Software
- Experience of the Software Industry
- Current Developments in the Software Industry





Student Training in Scientific Software Problems

- **Scientific computation** at universities is currently being driven by **procedural** techniques
- **Scientific computation** at universities is currently **implementation centric**, without following good **software engineering** practices of **requirements capture, analysis, design, testing, and implementation**
- Consequently, scientific software systems built by using practices are difficult to **maintain, extend, and reuse**. Such systems are difficult to decompose into **reusable components**
- This coding style leads to software systems which are **not robust under changes**, i.e. local changes often lead to global effects. This often leads to systems being **developed from scratch** whenever an old system needs to be modified or rebuilt, instead of **reusing** the **components** of already built systems



Problems (... continued)

- Students write code without considering the **needs of users** who are going to use the code in the future.
- New students can not understand this code and thus start **writing their own code from scratch**



Experience of the Software Industry

- Software industry has recognized the problem of **maintaining** and **modifying** systems based on functional decomposition, and has largely shifted towards building systems based on **object technology**.
- Experience gained by industry has lead to **processes** and **tools** for building object oriented systems, which are more **robust to changes**, and easier to **maintain** and **upgrade**.
- The movement towards building systems of applications based on **reusable components** is also showing a greater amount of **reuse** in the software industry.



Experience of the Software Industry (... continued)

- The promise of this technology was not realized overnight. There were **many false starts and failed projects**.
- The first generation of techniques did lead to code reuse, but also lead to systems, which were very **difficult to debug and modify**.
- Today, object technology is relatively mature and is using a **new generation** of techniques and processes, which are resulting into successful systems which are more **flexible**, and easier to **maintain, reuse** and **change**.
- **Simply using object technology, and a language which supports object orientation does not automatically lead to systems, which are robust and stable under changes.**



Current developments in the Software Industry

Patterns: Patterns are the result of a **grass root** movement, which originated in the **object software community**. The aim of this movement is to collect the **experience of successful software developers** in the form of patterns. (894U)

Iterative and Incremental Software Development Process: This is not a **top down** or **bottom up** approach. Instead, software is developed in well-defined **increments**. (894U)

Use Case Driven Processes: Modern processes start with modeling requirements in terms of **use cases**. All the activities of system building, i.e., analysis, design, implementation, and testing are **driven by use cases**. (894U)

Unified Modeling Language (UML): Recently, the **Object Management Group (OMG)** has standardized a modeling language, known as the **Unified Modeling Language (UML)**. (694T, 894U)

Current developments in the Software Industry (... continued)



Component Based Development: Components can be **reused**, and serve as **building blocks** to construct applications.

Standardized component models have been adopted by the industry, such as **Java Beans**, and **CORBA components** (694T)

Computer Aided Software Engineering (CASE) tools: Newer generation of CASE tools, such as **Rational Rose** from Rational Corporation, support the incremental and iterative process of system development quite well. (894U)

Distributed Computing: Industry has shifted away from **two tier** client-server models to **three tier** and **n-tier** systems. The client side is getting **thinner**, requiring less computational resources. The **business logic** is implemented using **EJB** (Enterprise Java Beans) on the middle tier, and middleware like **Application Servers**, **TMs** (Transaction Monitors), **OMs** (Object Monitors), **CORBA** (Common Object Request Broker Architecture), **RMI** (Remote Message Invocation), and **Jini**. These architectures **scale** well as the size of the system increases. (894V)



Current developments in the Software Industry (... continued)

Interoperability: Objects written in **different languages**, such as C++ and Java can communicate by using industry standard **CORBA** standardized by **OMG** (Object Management Group) or Java **JNI** (Java Native Interface) (894V)

Platform independence: Systems built by using **100% Java** can run on any modern operating system without the need for recompilation.

Mobile Code: Compiled **100% Java** byte code can execute on any operating system with a **JVM** (Java Virtual Machine) without a need for **recompilation**. Behavior of a remote application can be modified even after the application has been already deployed and activated. (694J)

XML (eXtensible Markup Language): This technology is causing a lot of excitement in the software industry. It presents the software community with a **standard** way to handle structure in data. (894V)

Current developments in the Software Industry (... continued)



Refactoring:

Refactoring is the process of changing a software system in such a way that it **does not alter the external behavior of the code** yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are “**improving the design of the code after it has been written.**”

That's an odd turn of phrase!

With refactoring you can take a bad design, chaos even, and rework it into well-designed code. Each step is simple, even simplistic. You **move a field from one class to another, pull some code out of a method to make into its own method, and push some code up or down a hierarchy.** Yet the cumulative effect of these small changes can **radically improve the design.** It is the exact reverse of the normal notion of software decay.

Quote from: “Refactoring : Improving the Design of Existing Code” 1999, Martin Fowler (Addison-Wesley Object Technology Series)