

Developing large-scale Applications in Python

*Lessons learned from 10 years of
Python Application Design*

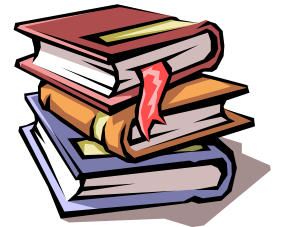
LSM Conference 2005
Dijon, France

Marc-André Lemburg

EGENIX.COM Software GmbH
Germany

Speaker Introduction: Marc-André Lemburg

- CEO eGenix.com and Consultant
 - More than 20 years software experience
 - Diploma in Mathematics
 - Expert in Python, OOP, Web Technologies and Unicode
 - Python Core Developer
 - Python Software Foundation Board Member (2002-04)
 - Contact: mal@egenix.com
- eGenix.com Software GmbH, Germany
 - Founded in 2000
 - Core business:
 - **Consulting**: helping companies write successful Python software
 - **Product design**: professional quality Python/Zope developer tools (mxODBC, mxDateTime, mxTextTools, etc.)
 - International customer base



Agenda

1. Introduction
2. Application Design
3. Before you start...
4. Discussion



Introduction

1. Introduction
2. Application Design
3. Before you start...
4. Discussion



Designing Python Applications

- Python makes it very **easy to write complex applications** with very little code
 - It's easy to **create bad designs fast**
 - **Rewriting code is fast** as well
- **Application design** becomes the most important factor in Python projects
- This talk presents a **general approach** to the problem
 - ... which is not necessarily specific to Python

Large-scale applications

- What is considered “**large-scale**” in Python ?
 - Server application: **>30 thousand lines** of Python code
 - Client application: **>10 thousand lines** of Python code
 - Third-Party code: >10 thousand lines of code
 - Typically a mix of Python code and C extensions
- Examples:
 - Zope / Plone
 - eGenix Application Server
 - eGenix projects: e.g. Web Service Application Server, XML Database, ASP Trading System

Why write large-scale applications in Python ?

- Highly **efficient**
 - small teams can scale up against large companies
 - very competitive turn-around times
 - small investments can result in high gains
- Very **flexible**
 - allows **rapid design, refactoring and rollout**
 - highly adaptive to new requirements and environments
 - no lock-in
- Time-to-market
 - develop in weeks rather than months

Application Design

1. Introduction
2. Application Design
3. Before you start...
4. Discussion



The Design Concept

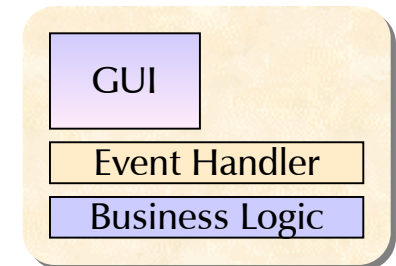
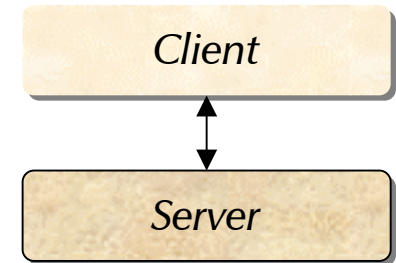
- Structured approach to application design
 - *Divide et Impera* (divide and conquer)
 - Lots and lots of experience ☺
- The Zen in application design
 - Keep things as simple as possible, but not simpler (KISS).
 - There's beauty in design.
 - Before doing things twice, think twice.
 - If things start to pile up, management is needed.
 - If management doesn't help, decomposition is needed.

Divide et Impera: Steps in the design

- Goal: Break down complexity as far as possible !
- Top-down method:
 1. Application model
 2. Processing model
 3. Layer model
 4. Components
 5. Management objects
 6. Data and Task objects

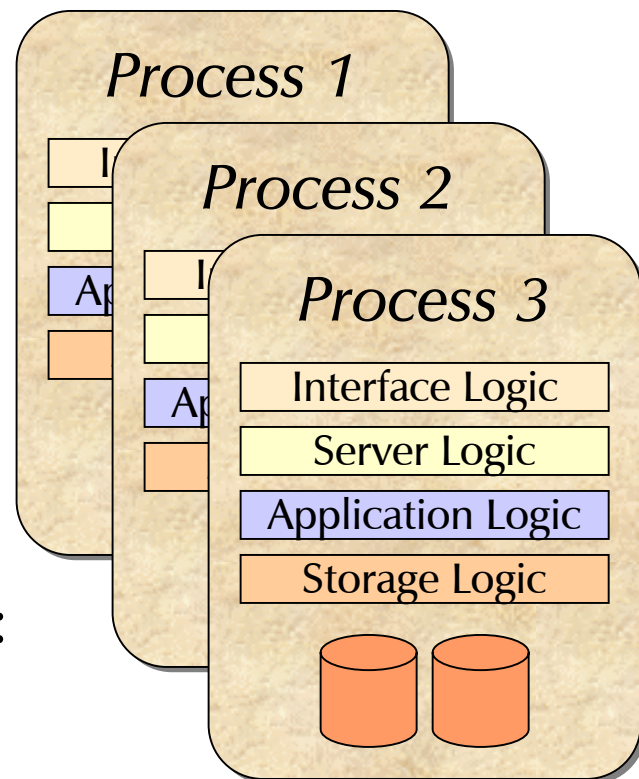
Choose a suitable *application model*

- Client-Server
 - Client application / Server application
 - Web client / Server application
- Multi-threaded stand-alone
 - Stand-alone GUI application
- Single process
 - Command-line application
 - Batch job application
- etc.



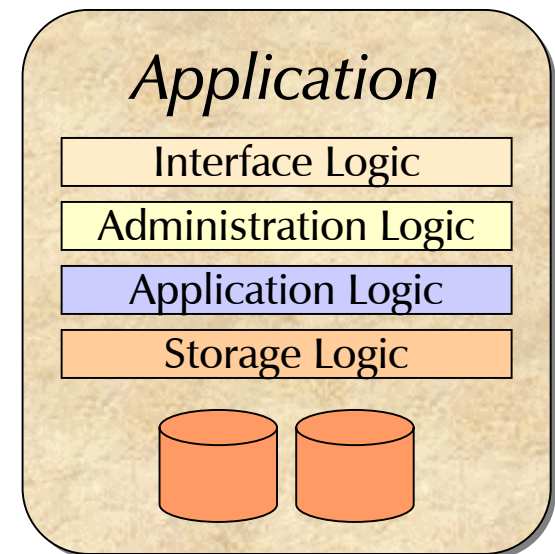
Identify the *processing model*

- Identify the **processing scheme**:
 - Single process
 - Multiple processes
 - Multiple threads
 - Asynchronous processing
 - A mix of the above
- Identify the **process/thread boundaries**:
 - Which components (need to) share the same object space ?
 - Where is state kept ?
 - What defines an application instance ?



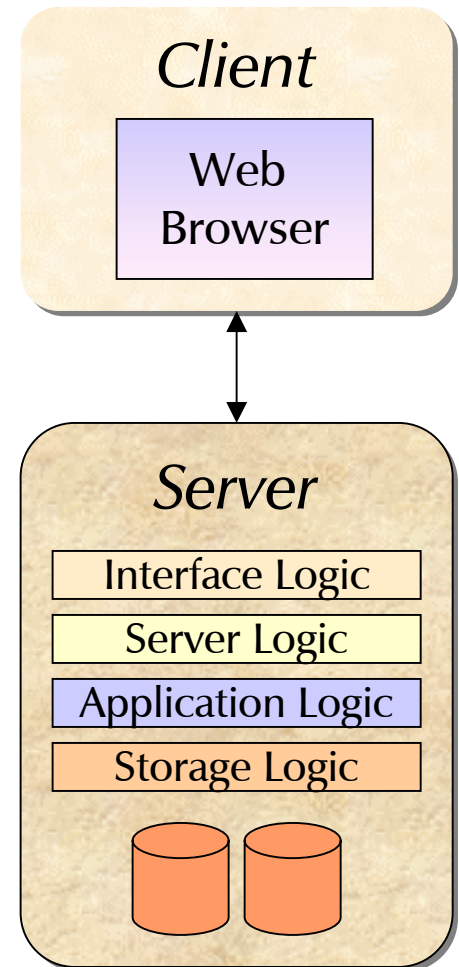
Layer model

- Every application can be divided into **layers of functionality** defined by the flow of data through the application
 - **Top layer:** interface to the outside world
 - **Intermediate layers:** administration and processing
 - **Bottom layer:** data storage
- Layers are usually easy to identify given the application model
 - ... but often hard to design



Examples of layer models

- GUI / Application Logic / Storage Logic
- Network / Apache / SCGI / Server Logic / Application Logic / Storage Logic
- File I/O / Application Logic / Storage Logic
- Custom model

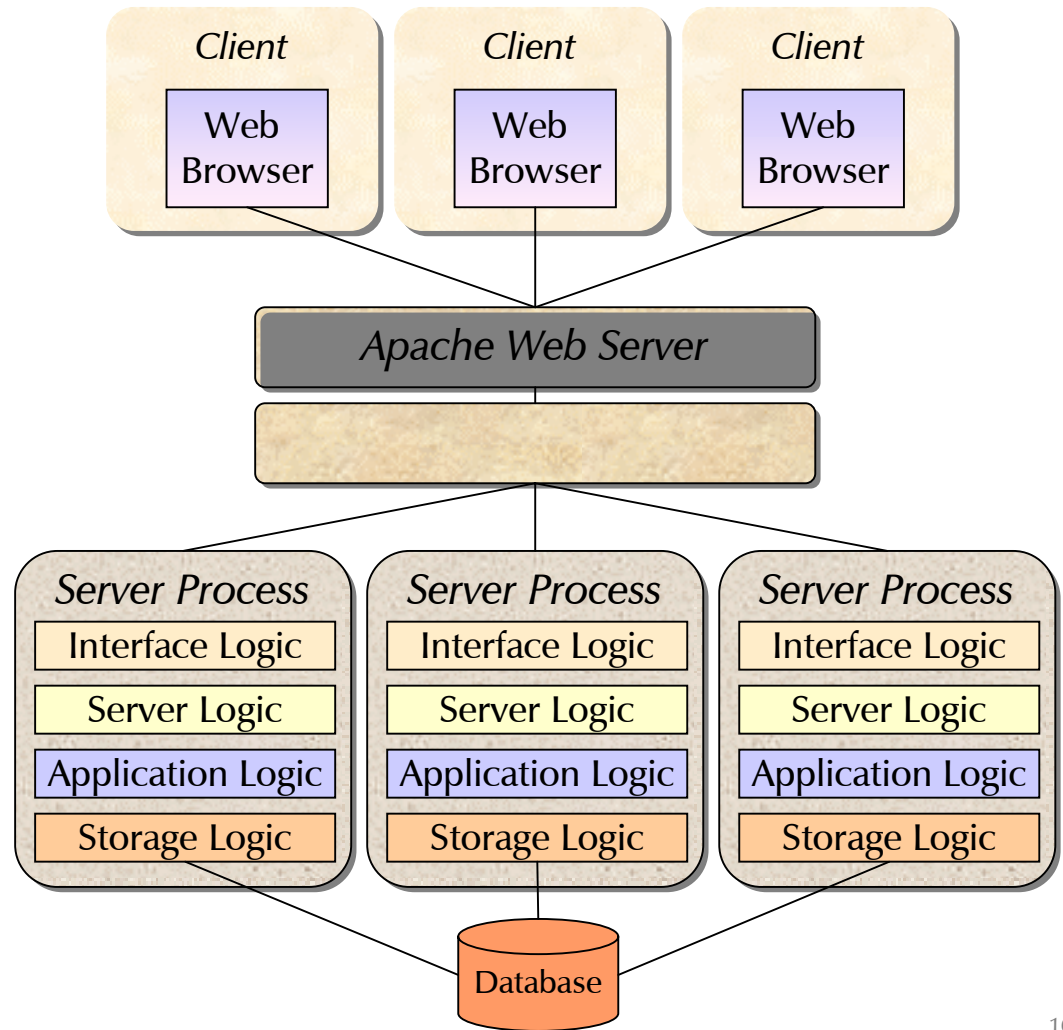


Example Web Client + Server Application: Problem

- Situation:
 - Client is a standard [web-browser](#)
 - Server needs to take a lot of load and will have to do all the calculation work
 - Server needs to be [fail-safe](#)
 - Server is connected to a database

Example Web Client + Server Application: Solution

- Solution:
 - Application model: client-server
 - Processing model: multiple process model
 - Layer model: typical application server layers



Divide et Impera: Layers are still too complex

- Goal: Break down complexity as far as possible !
- Top-down approach:
 1. Application model
 2. Processing model
 3. Layer model
 4. Components
 5. Management objects
 6. Data and Task objects

Breaking layers into smaller pieces: *Components*

- Layers provide a data driven separation of functionality
- Problem:
 - The **level of complexity is usually too high** to implement these in one piece of code
- Solution:
 - build layers using a set of **loosely coupled components**



Component design

- Components should encapsulate **higher level concepts** within the application
- Components provide **independent building blocks** for the application

Component examples

- Components ...
 - provide the **database** interface
 - implement the **user** management
 - implement the **session** management
 - provide **caching** facilities
 - interface to **external data** sources
 - provide **error handling** facilities
 - enable **logging** management
 - etc.



Advantages of components: Easily replaceable

- They should be **easily replaceable** to adapt the application to new requirements, e.g.
 - porting the application to a new database backend,
 - using a new authentication mechanism, etc.

- If implemented correctly, they will even allow **switching to different processing model** should the need arise

Advantages of components: Loose coupling

- Loose coupling of the components makes it possible to
 - refine the overall application design,
 - refactor parts of the layer logic or
 - add new layers
- without having to rewrite large parts of the application code

Component implementation

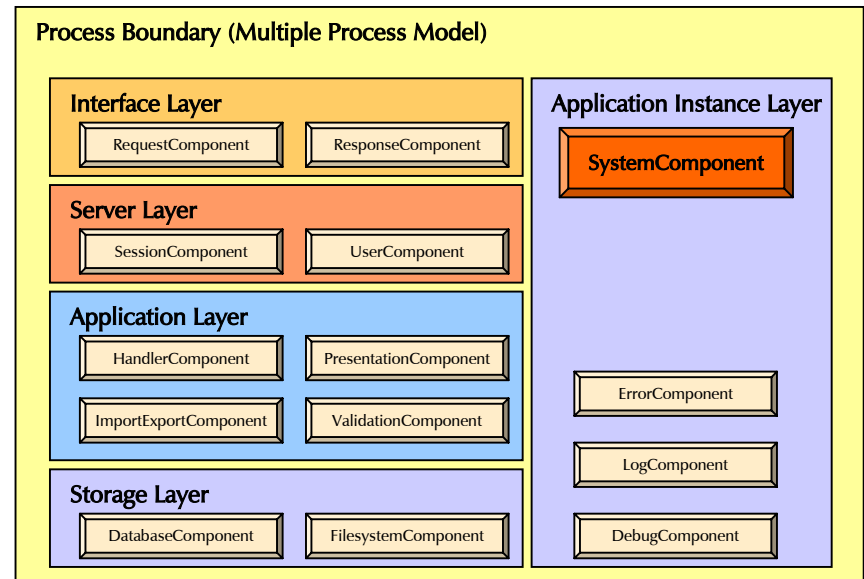
- Each component is represented by a **component object**
- Component interfaces must be **simple and high-level** enough to allow for **loose coupling**
 - Internal parts of the components are never accessed directly, only via the component interface
- Component objects should **never keep state** across requests
 - Ideally, they should be thread-safe

Special component: The *System Object*

- One *system component object* which represents the application instance
 - All component objects are created and managed by the system object
 - Components can access each other through the system object (circular references !)
 - There can be multiple system objects, e.g. one running in each thread

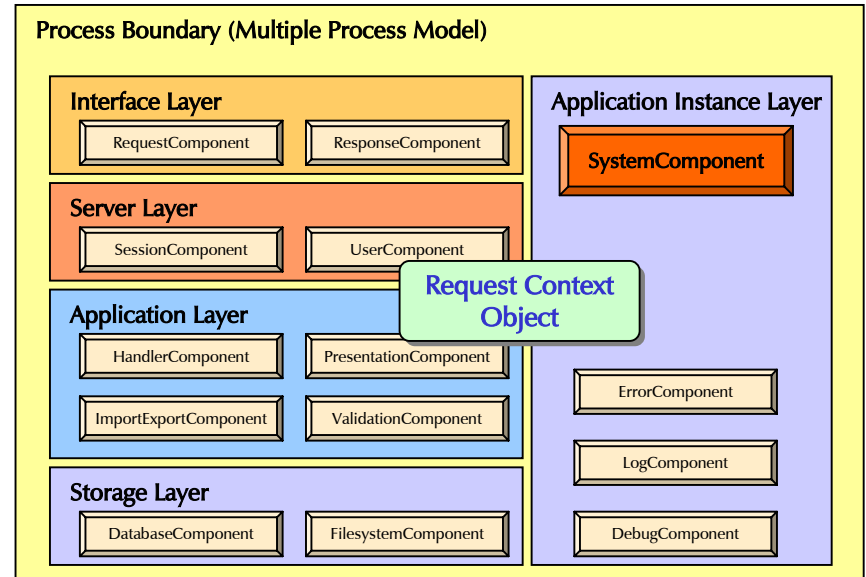
Split the layers into components

- General approach:
 - One system component that manages the application instance
 - At least one component per layer



Managing data: The *Request Context Object*

- Data management:
 - Global data is only used for configuration purposes
 - Components **don't store per-request state** !
 - Per-request global data is stored and passed around via **Request Context Objects**



Layers and components: The big picture

Process Boundary (Multiple Process Model)

Interface Layer

RequestComponent

ResponseComponent

Server Layer

SessionComponent

UserComponent

Application Layer

HandlerComponent

PresentationComponent

ImportExportComponent

ValidationComponent

Storage Layer

DatabaseComponent

FilesystemComponent

Application Instance Layer

SystemComponent

All Component Objects
are connected to the
SystemComponent
object

ErrorComponent

LogComponent

DebugComponent

Reduce component complexity: *Management objects*

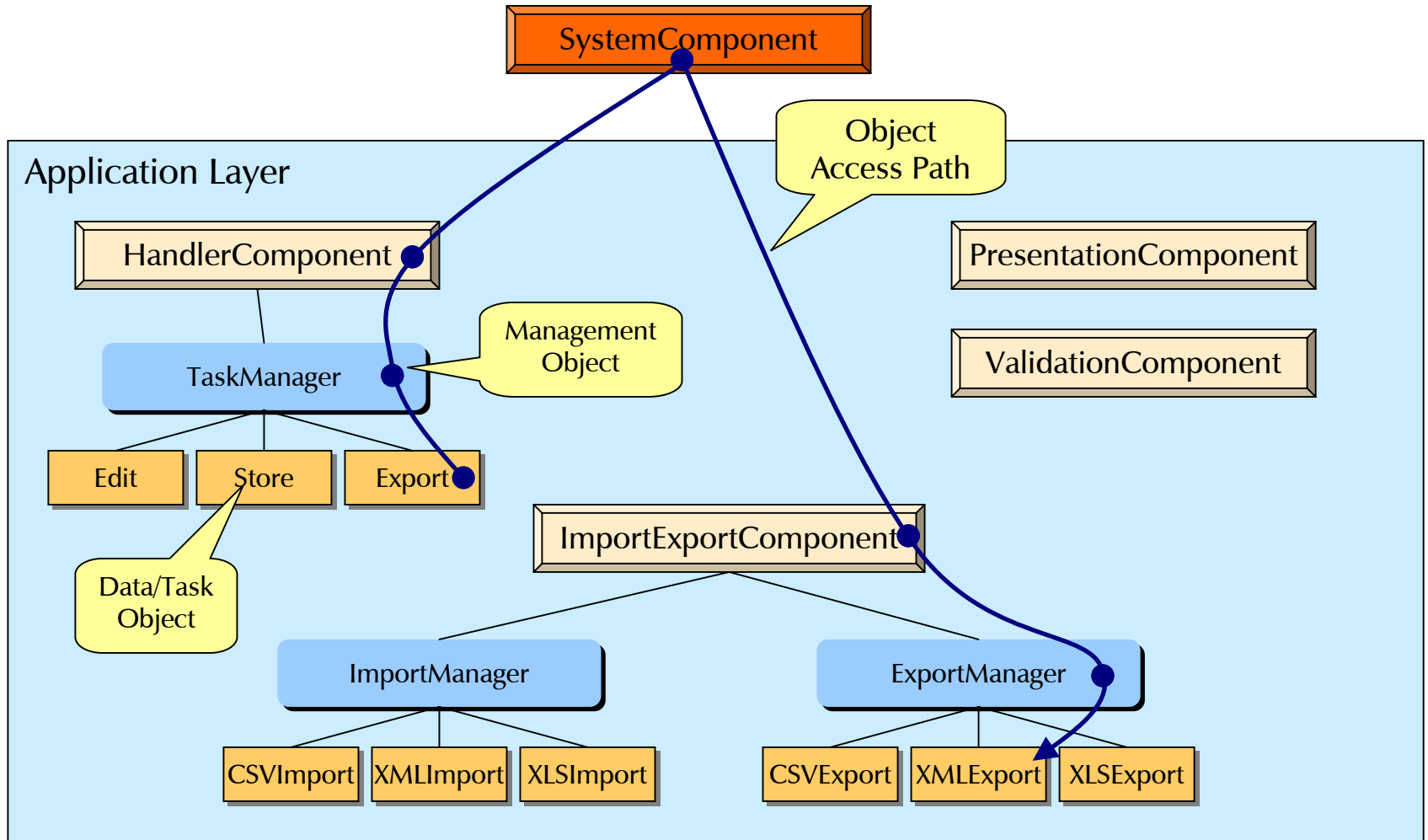
- **Management objects**
 - help **simplify component object implementations**
 - work on or with groups of low **data/task objects**

- The distinction between management objects and component objects is not always clear
 - If in doubt, use a component object that delegates work to a management object

Low-level: *Data and task objects*

- Data objects
 - encapsulate data (nothing much new here)
- Task objects
 - interaction with multiple objects
 - I/O on collections of objects
 - delegating work to other management objects
 - interfacing to component objects
 - etc.

Example: Management objects at work



Management object or component ?

- Use **component objects** to represent logical units / concepts within the application
 - without going into too much detail...
- Use **management objects** to work on collections of data/task objects
 - to simplify component implementations
 - to avoid direct interfacing between the data/task objects

➤ **Never mix responsibilities**



Intermezzo: There's beauty in design



Before you start...

1. Introduction
2. Application Design
3. Before you start...
4. Discussion



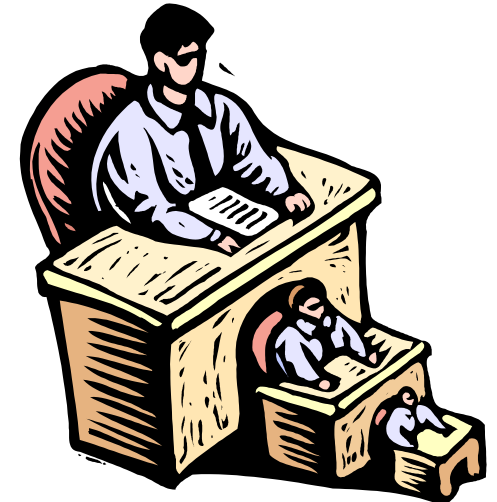
Structuring your modules

- First some notes on the import statement:
 - Keep **import dependencies low**;
avoid “from ... import *”
 - Always **use absolute import paths**
(defeats pickle problems among other things)
 - Always layout your application modules **using Python packages**
 - Import loops can be nasty;
import on demand can sometimes help



Finding the right package structure

- Use **one module** per
 - management/component class
 - group of object classes managed by the same management class
 - keep modules small; if in doubt, split at class boundaries
- Group components and associated management modules in **Python packages** (directories)
- Use the application model as basis for the package layout



Data, classes and methods

- Use **data objects** for data encapsulation...
 - instead of simple types (tuples, lists, dictionaries, etc.)
- Use **methods** even for simple tasks...
 - but don't make them too simple
- Use **method groups** for more complex tasks
 - e.g. to implement a storage query interface
- Use **mix-in classes** if method groups can be deployed in more than class context
 - If you need to write the same logic twice, think about creating a mix-in class to encapsulate it, or put it on a base class
 - Avoid using mix-in classes if only one class makes use of them



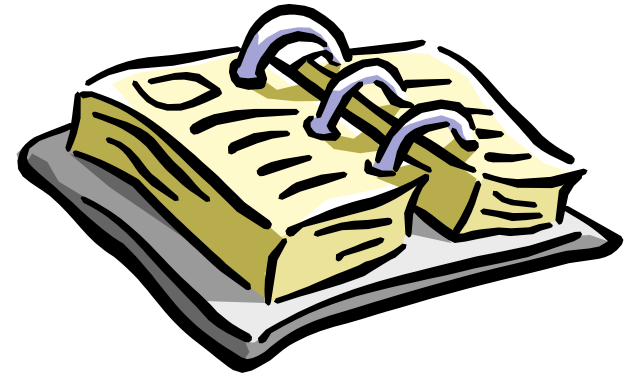
Make mistakes... and learn from them: *Refactoring*

- If an **implementation gets too complicated**, sit down and reconsider the design...
 - often enough a small change in the way objects interact can do wonders
- **Be daring when it comes to rewriting larger parts of code !**
 - It sometimes takes more than just a few changes to get a design right
 - It is often faster to implement a good design from scratch, than trying to fix a broken one



Often forgotten: Documentation

- Always document the code that you write !
- Use doc-strings, inline comments and block logical units using empty lines...
 - doc-strings represent your method's contracts with the outside world
- Document the intent of the methods, classes and logical code units...
 - not only their interface
- Use descriptive identifier names...
 - even if they take longer to type



Quality Assurance: XP helps !

- Use **extreme programming techniques** whenever possible:
 - Always **read the code** top to bottom after you have made changes or added something new to it
 - Try to follow the flow of information in your mind (before actually running the code)
 - **Write unit tests** for the code and/or test it until everything works as advertised in the doc-strings



Quality Assurance: A few tips

- **Typos can easily go unnoticed in Python:**
use the editor's auto-completion function as often as possible
- Use tools like **PyChecker** to find hidden typos and possibly bugs
- Always test code **before committing** it to the software repository



Discussion

1. Introduction
2. Application Design
3. Before you start...
4. Discussion



Conclusion

- Structured application design can go a long way
- Divide-et-impera helps keep basic buildings blocks manageable
- Extreme programming doesn't have to spoil the fun

All this sounds familiar...

- Application design is in many ways like **structuring a company**:
 - Divisions need to be set up (component objects)
 - Responsibilities need to be defined (management vs. data/task objects)
 - Processes need to be defined (component/management object APIs)
- Applications **work** in many ways **like companies**:
 - Customer interaction (user interface)
 - Information flow (application interface)
 - Decision process (business logic)
 - Accounting and data keeping (storage interface)

Developing large-scale applications in Python

- Questions

- Has anyone worked on large-scale Python applications ?
- What tools / features are (still) missing in the tool chain ?
- Would you be prepared to pay for components or frameworks ?



And finally...



Thank you for your time.

Contact

eGenix.com Software, Skills and Services GmbH

Marc-André Lemburg

Pastor-Löh-Str. 48

D-40764 Langenfeld

Germany

eMail: mal@egenix.com

Phone: +49 211 9304112

Fax: +49 211 3005250

Web: <http://www.egenix.com/>