

Designing large-scale applications in Python

**EuroPython 2013
Florence, Italy**

Marc-André Lemburg

Marc-André Lemburg

EGENIX.COM™



Agenda

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

Agenda

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

Designing Python Applications (1/2)

- 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

Designing Python Applications (2/2)

- **Application design** becomes the most important factor in Python projects
- This talk presents a **general approach** to the problem

Large-scale applications

- What can be considered “large-scale” in Python ?
 - Server application:
 - > 100 thousand lines of Python code
 - Client application:
 - > 50 thousand lines of Python code
 - Third-Party code:
 - > 100 thousand lines of code
 - Typically a mix of Python code and C extensions

Why write applications in Python ? (1/3)

- **Highly efficient**
 - Small teams can scale up against large companies
 - Very competitive turn-around times
 - Small investments can result in high gains

Why write applications in Python ? (2/3)

- **Very flexible**
 - allows rapid design, refactoring and rollout
 - highly adaptive to new requirements and environments
 - no lock-in

Why write applications in Python ? (3/3)

- Time-to-market
 - Develop / add new features
in weeks rather than months
 - Be ahead of the game or
 - Stay competitive

Agenda

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

Situation

- A typical application scenario:
 - Complex interactions between program parts
 - Complex work concepts
 - Many different I/O types

High-level view

- Applications typically have to implement...
 - Customer interaction ([user interface](#))
 - Information flow ([application interface](#))
 - Decision process ([business logic](#))
 - Accounting and data keeping ([storage interface](#))

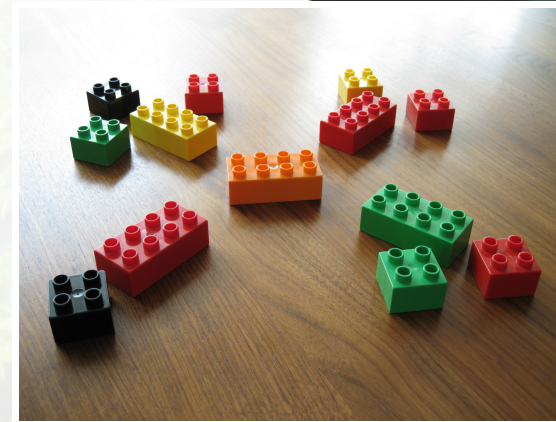
Think outside the box...

- Application design is in many ways like **structuring a company**:
 - Departments and divisions need to be set up
 - Responsibilities need to be defined
 - Processes need to be defined

The Design Concept

(1/2)

- **Structured approach** to application design
 - **Divide et Impera** (divide and conquer)
 - Top-down method:
 - Application model
 - Processing model
 - Layer model
 - Components
 - Management objects
 - Data and Task objects
- **Lots of experience** also helps...



The Design Concept

(2/2)

- Zen of Application Design (from this import app)

- Keep things as simple as possible, but not simpler (KISS)
- Before doing it, think twice
- If things get too complex, more is needed.
- If management doesn't help, composition is needed.
- Keep in mind: There's beauty in design.

from this import app...



The Design Concept

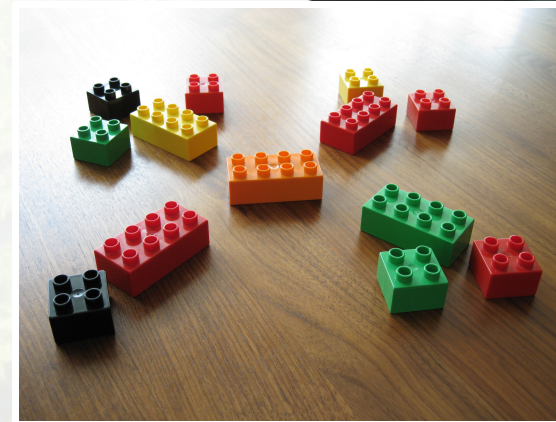
(2/2)

- **Zen of Application Design** (from this import app)
 - Keep things as simple as possible, but not simpler (**KISS**).
 - Before doing things twice, think twice (**DRY**).
 - If things start to get too complex, **decomposition** is needed.
 - If decomposition doesn't help, **management** is needed.
 - Keep in mind: There's **beauty in design**.



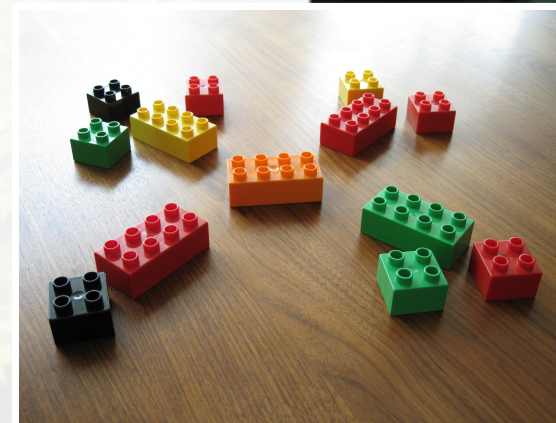
Divide et Impera: Step by step

- Goal: Break down complexity as far as possible !
- Top-down method:
 - Application model
 - Processing model
 - Layer model
 - Components
 - Management objects
 - Data and Task objects



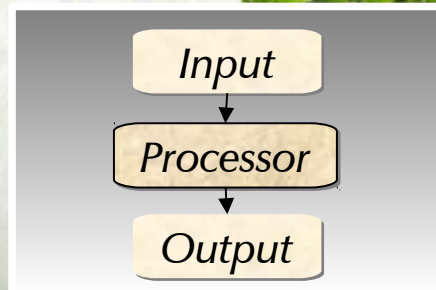
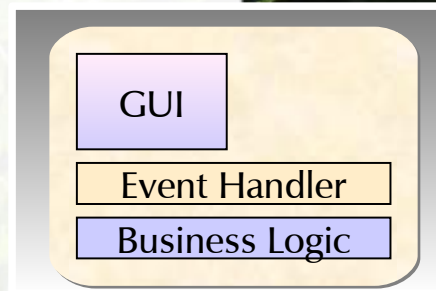
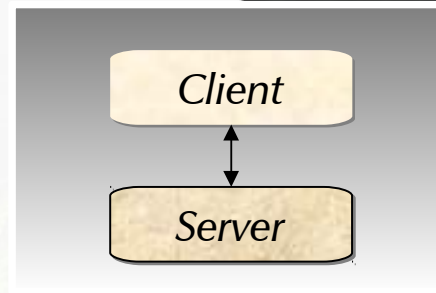
Start with the type of application

- Goal: Break down complexity as far as possible !
- Top-down method:
 - Application model
 - Processing model
 - Layer model
 - Components
 - Management objects
 - Data and Task objects



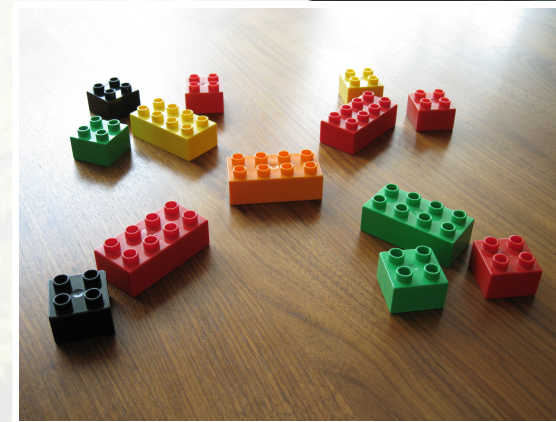
Choose a suitable *application model*

- 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
- ...



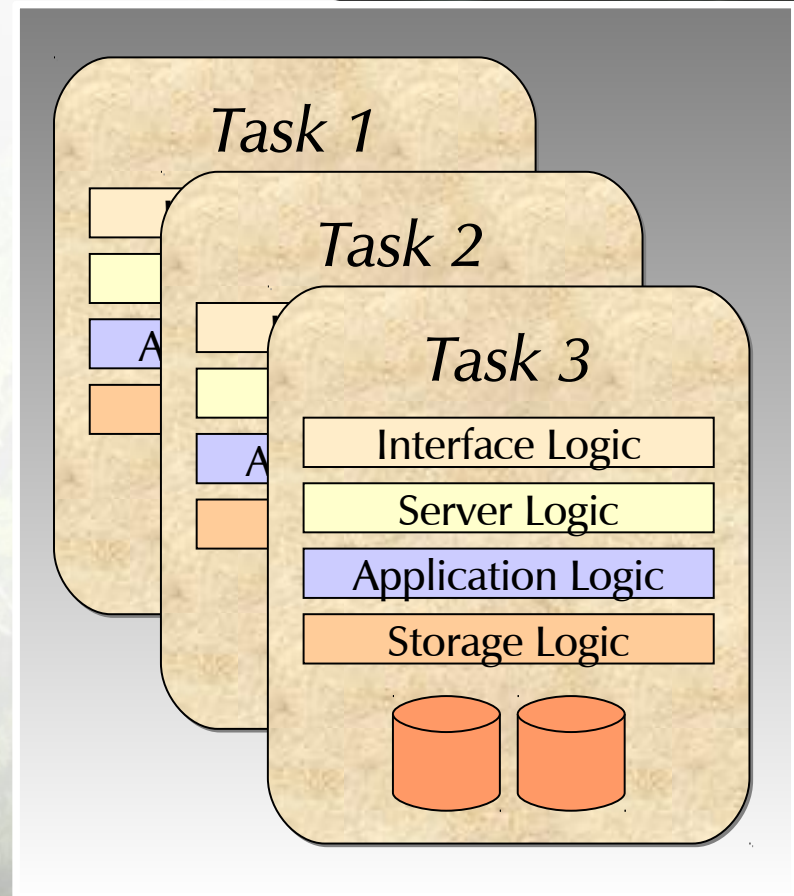
How should requests be processed ?

- Goal: Break down complexity as far as possible !
- Top-down method:
 - Application model
 - **Processing model**
 - Layer model
 - Components
 - Management objects
 - Data and Task objects



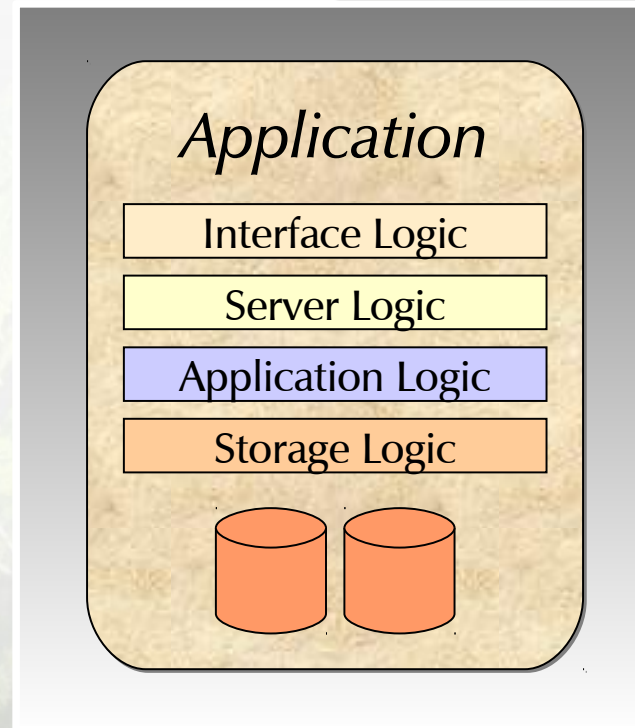
Identify the *processing model*

- Identify the **processing scheme**:
 - Single process
 - Multiple processes
 - Multiple threads
 - Asynchronous processing
 - A mix of the above



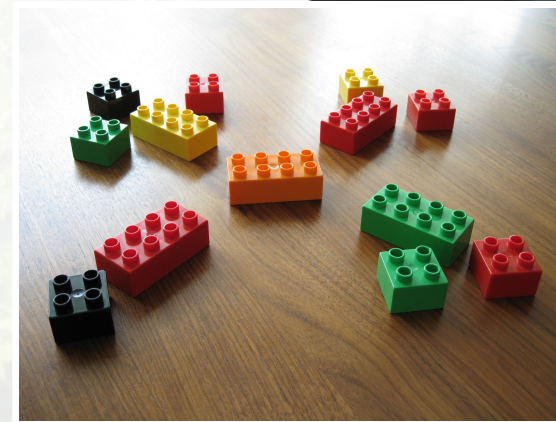
Identify the *processing model*

- Identify the **process/thread boundaries**:
 - Which components (need to) share the same object space ?
 - Where is state kept ?
 - What defines an application instance ?



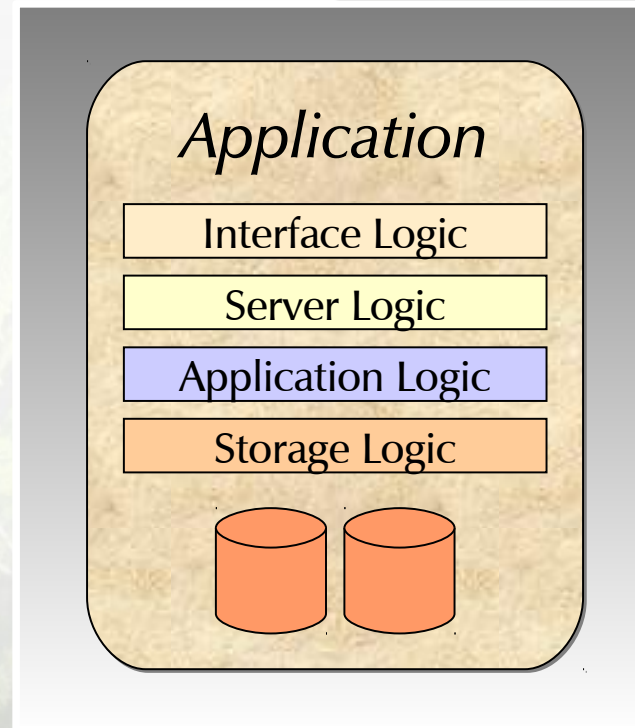
Break down by functionality

- Goal: Break down complexity as far as possible !
- Top-down method:
 - Application model
 - Processing model
 - **Layer model**
 - Components
 - Management objects
 - Data and Task objects



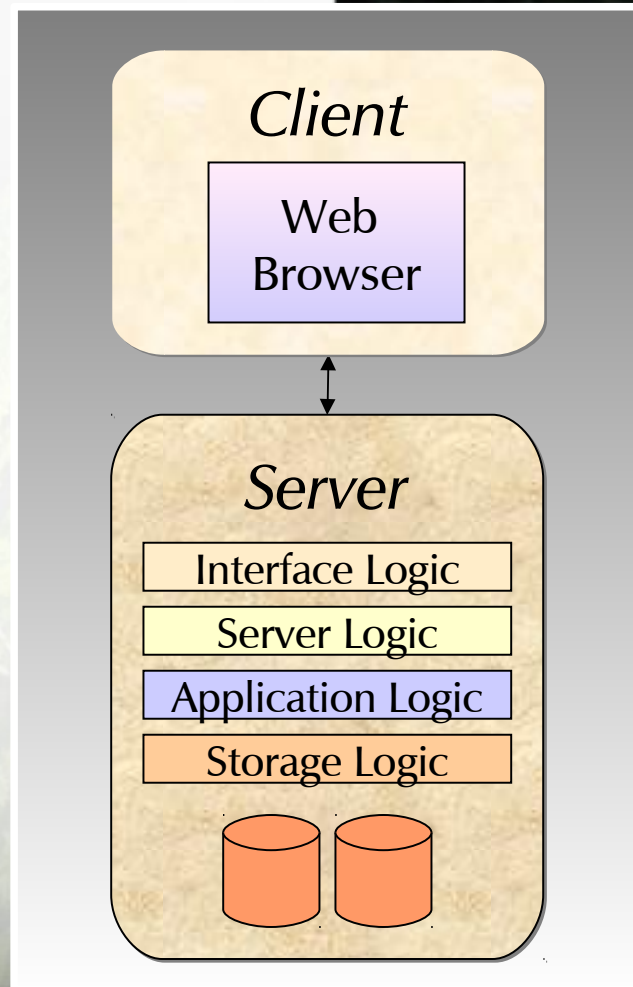
Find the right *layer model*

- 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



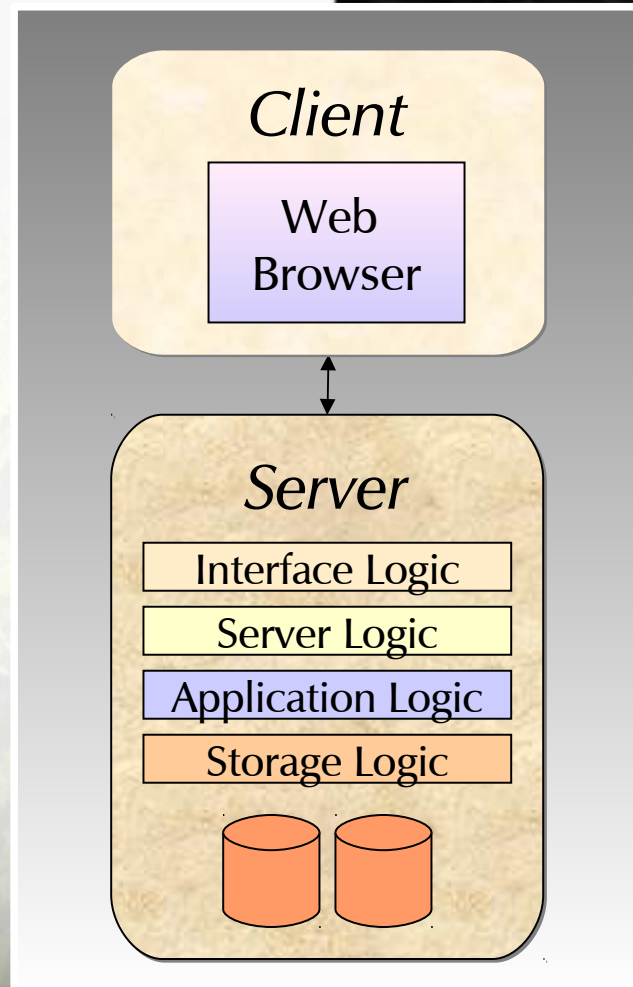
Examples of layer models

- **Client application:**
GUI / Application Logic / Storage Logic
- **Web application:**
Web Browser/ Network / Web Server /
Interface Logic (SCGI, WSGI) /
Server Logic / Application Logic /
Storage Logic
- **Batch processing:**
File I/O / Application Logic /
Storage Logic
- Custom model



Examples of layer models

- Client application:
GUI / Application Logic / Storage Logic
- **Web application:**
Web Browser/ Network / Web Server /
Interface Logic (SCGI, WSGI) /
Server Logic / Application Logic /
Storage Logic
- Batch processing:
File I/O / Application Logic /
Storage Logic
- Custom model

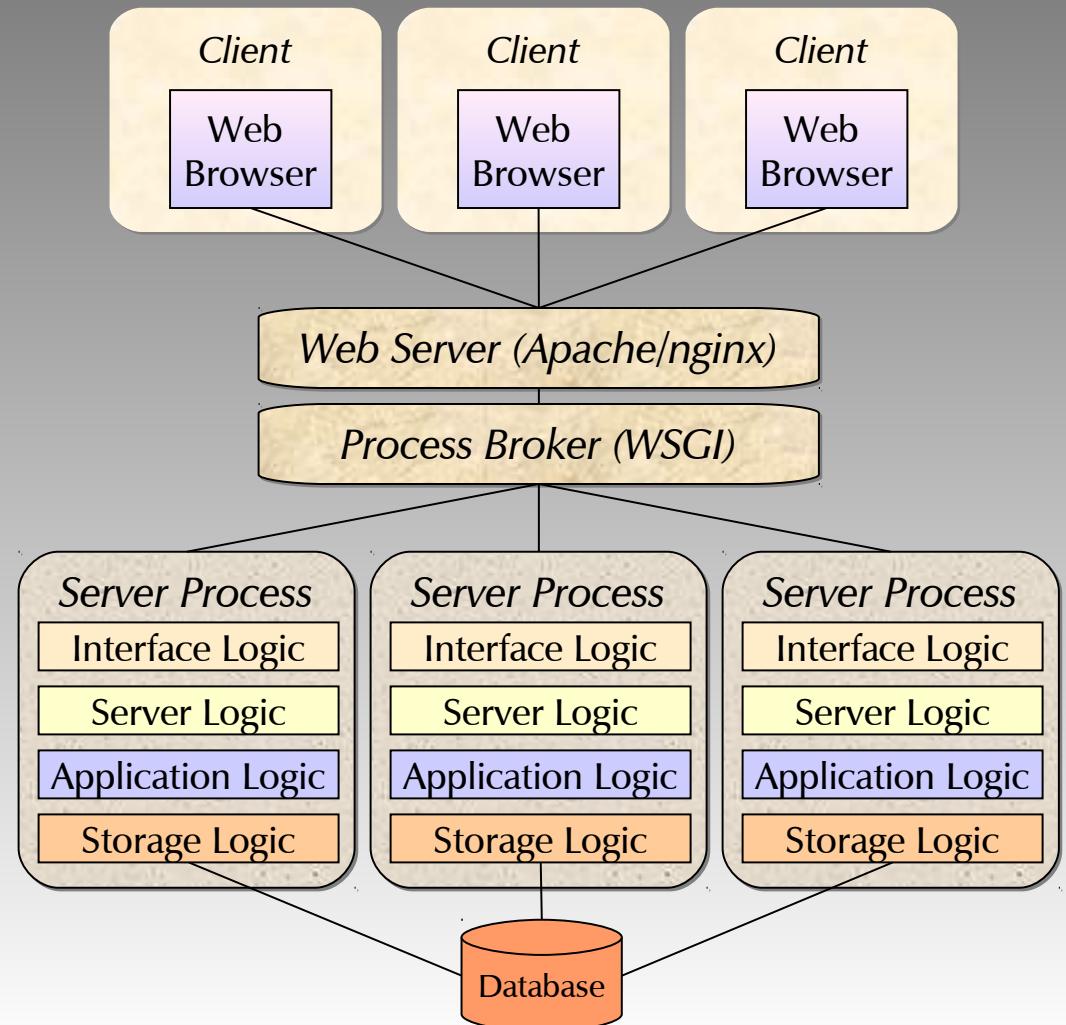


Example: Web Application

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

Example: Web Application

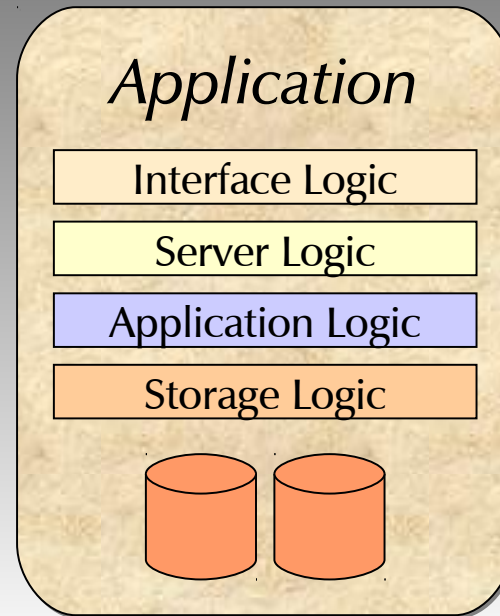
- Solution:
 - Application model:
client-server
 - Processing model:
multiple process
model
 - Layer model:
typical application
server layers



Found the *layer model*: now what... ?

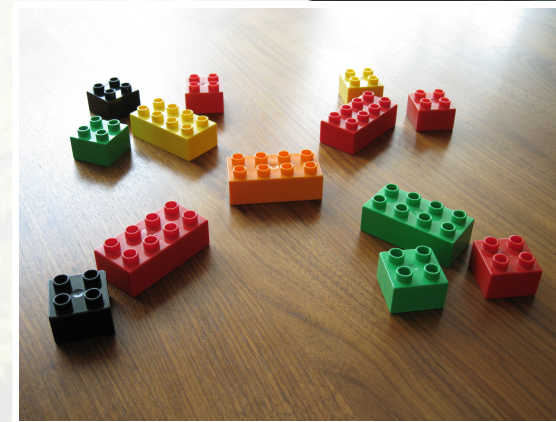
- Layers are usually easy to identify, given the application model

... but often hard to design



Layers are still too complex

- Goal: Break down complexity as far as possible !
- Top-down approach:
 - Application model
 - Processing model
 - Layer model
 - **Components**
 - Management objects
 - Data and Task objects



Break up layers into *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

(1/2)

- They should be **easily replaceable** to adapt the application to new requirements, e.g.
 - porting to a new database backend,
 - using a new authentication mechanism, etc.
- If implemented correctly, they will even allow **switching to a different processing model**, should the need arise.

Advantages of components

(2/2)

- **Loose coupling** of the components makes it possible to
 - refine the overall application design,
 - refactor parts of the layer logic, or
 - add new layers

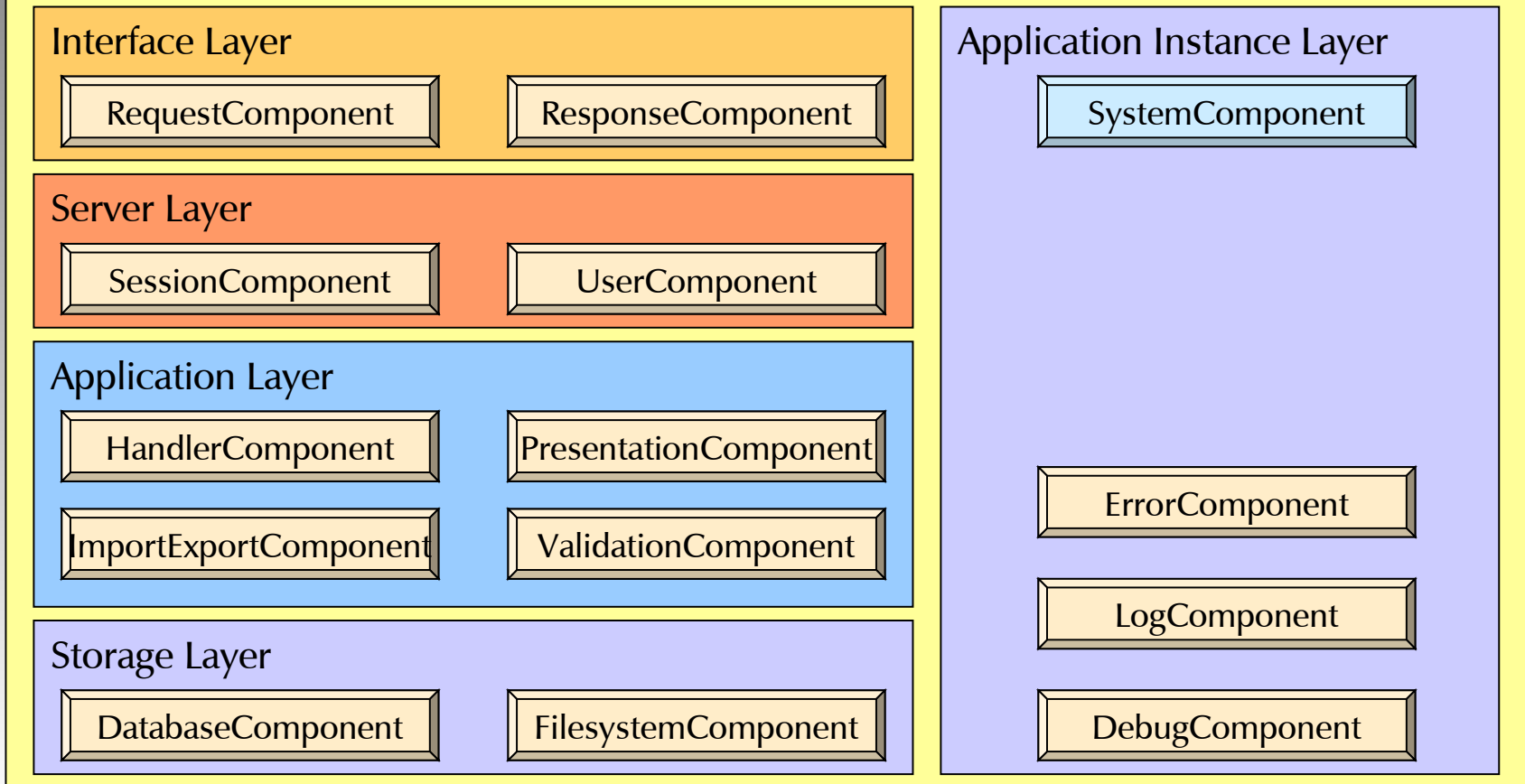
without having to rewrite large parts of the application code

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 also be thread-safe

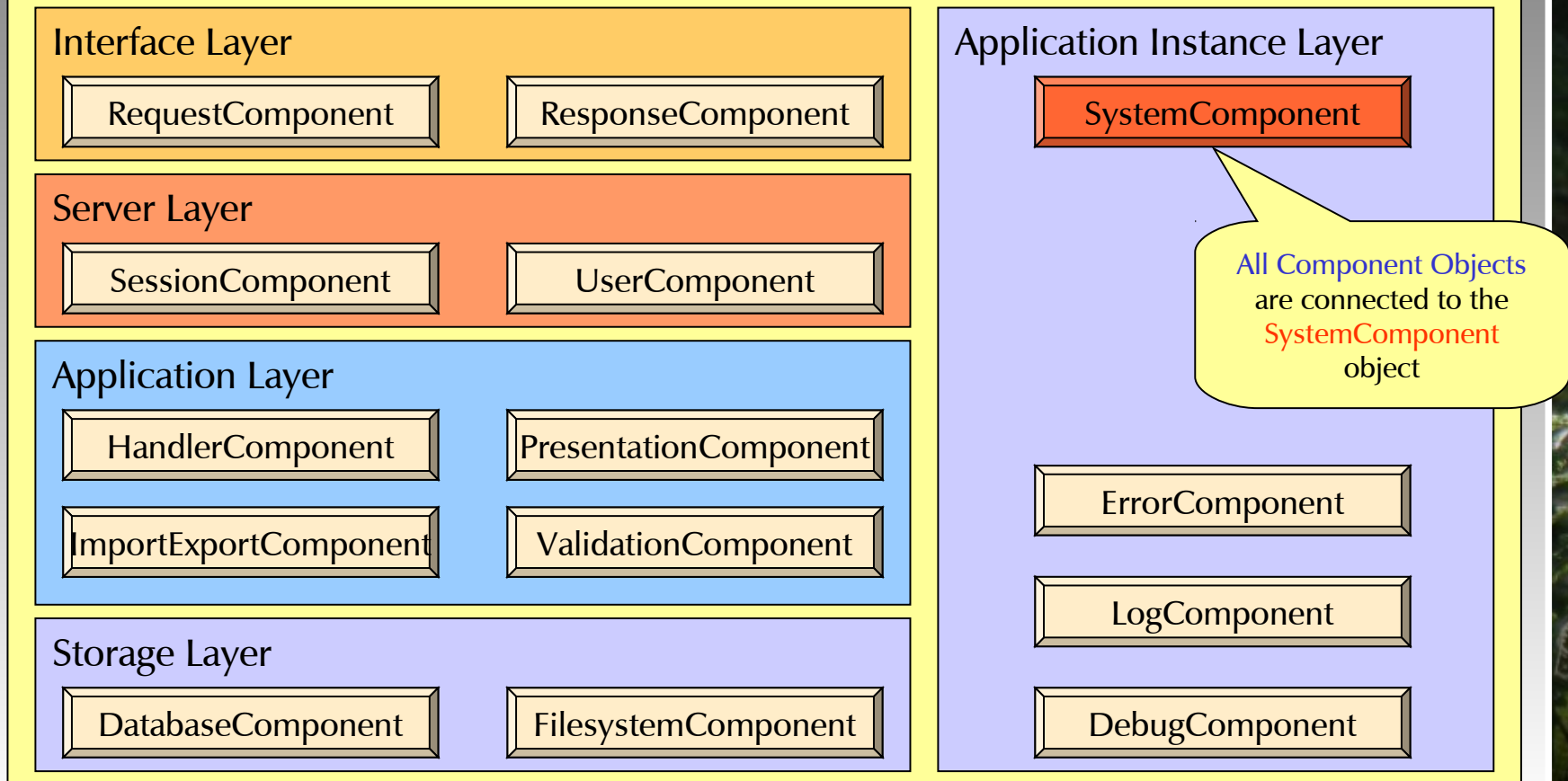
The Big Picture

Process Boundary (Multiple Process Model)



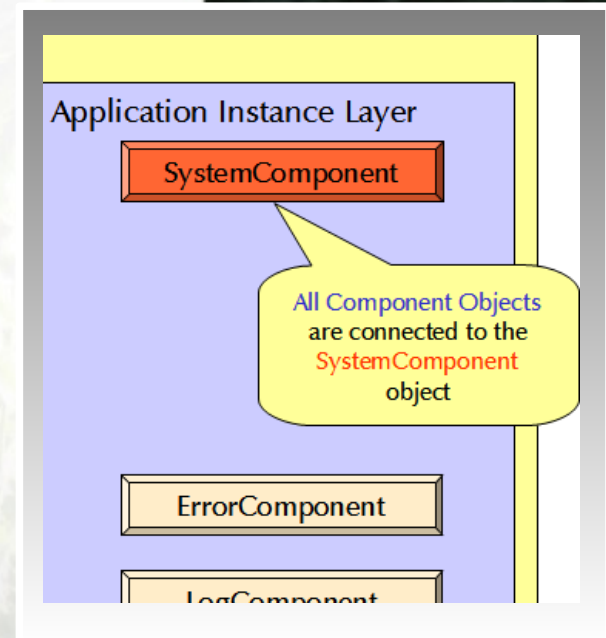
The Big Picture

Process Boundary (Multiple Process Model)



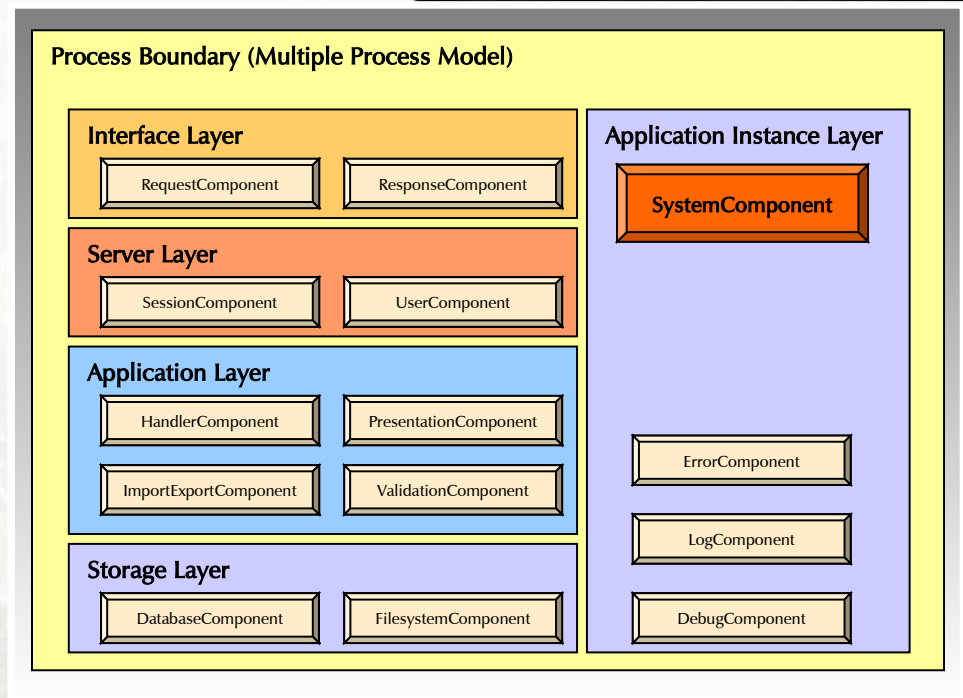
The special *System Object*

- 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



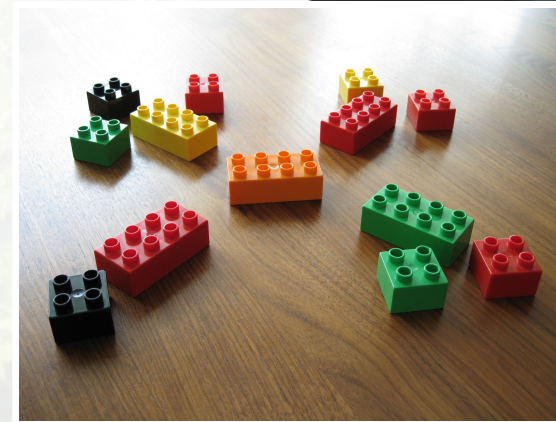
Components: Summary

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



Components too complex as well ?

- Goal: Break down complexity as far as possible !
- Top-down approach:
 - Application model
 - Processing model
 - Layer model
 - Components
 - **Management objects**
 - Data and Task objects



Add *management objects*

- **Management objects**
 - help simplify component object implementations
 - work on or with groups of low-level data/task objects
 - provide application internal APIs
 - interface to the “outside world”, e.g. file system, database, GUI, etc.

Note:

The distinction between management objects and component objects is not always clear ...

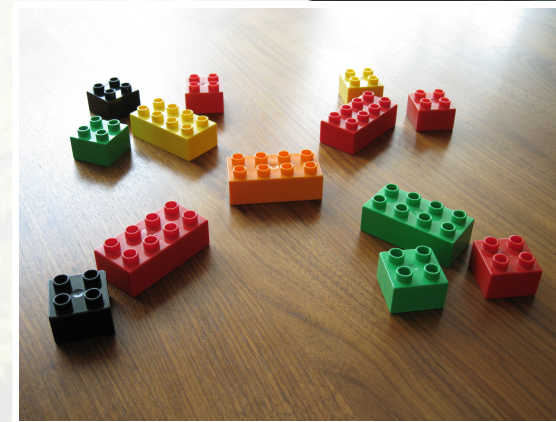
Management object or component ?

- 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

Try to never mix responsibilities

Divide et Impera: The Lowest Level

- Goal: Break down complexity as far as possible !
- Top-down approach:
 - Application model
 - Processing model
 - Layer model
 - Components
 - Management objects
 - **Data and Task objects**



Lowest level: *Data and task objects*

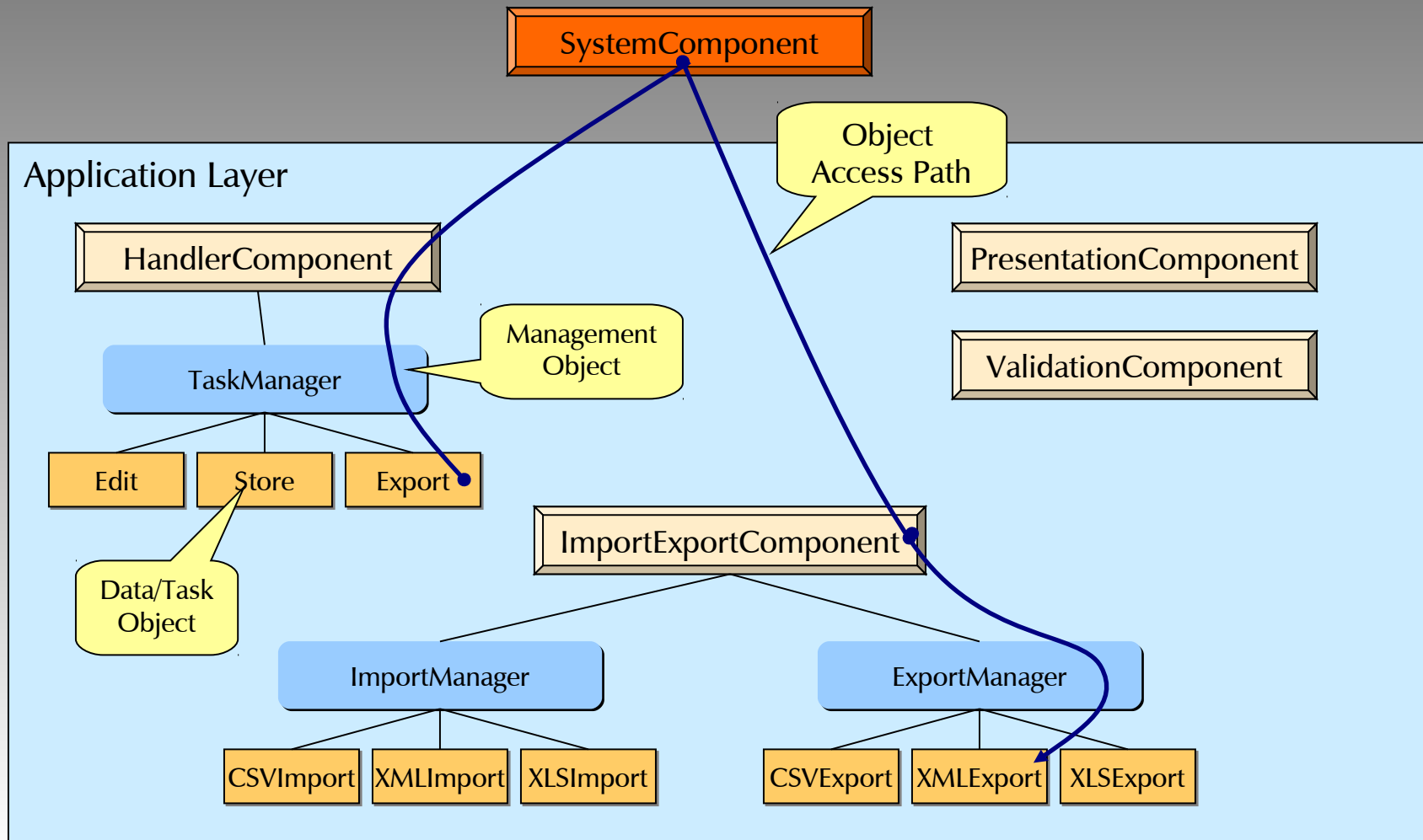
Data objects

- encapsulate data (nothing much new here)

Task objects

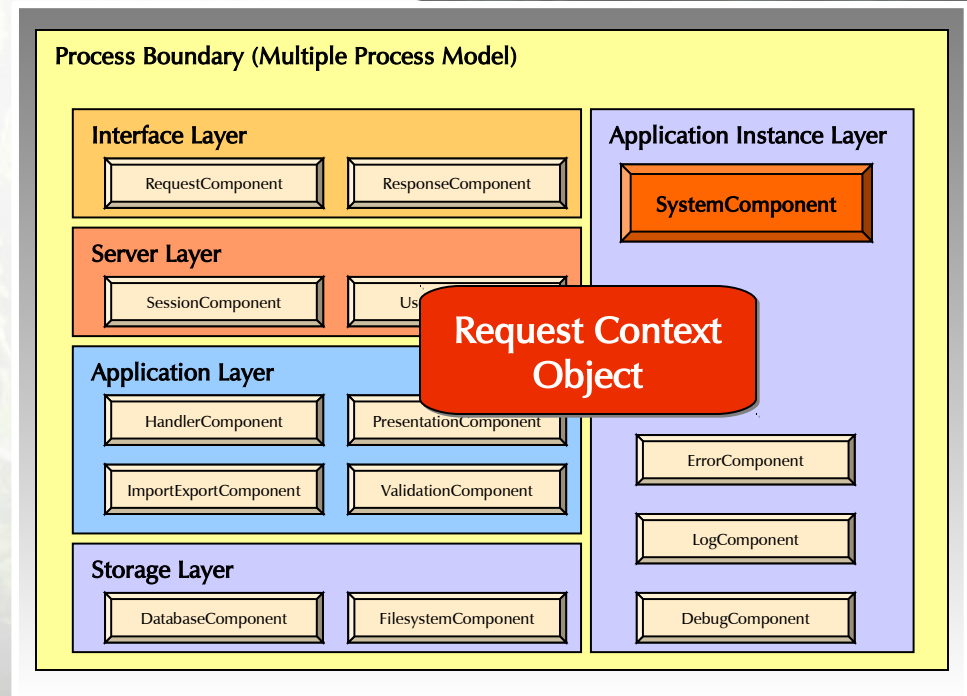
- interaction with multiple objects
- I/O on collections of objects
- delegating work to other management objects
- interfacing to component objects
- etc.

Example: Internal Communication



The *Request Context Data Object*

- This is useful for task based applications, e.g. web applications
- Data management:
 - Components **don't store per-request state** !
 - Per-request data is stored and passed around via **Request Context Objects**



And don't forget: There's beauty in design !



Thinking outside the box... a recap

- Application design is in many ways like **structuring a company**:
 - Departments and divisions need to be set up (**layer** and **component objects**)
 - Responsibilities need to be defined (**management** vs. **data/task objects**)
 - Processes need to be defined (**component/management object APIs**)

Conclusion

- Structured application design can go a long way
- Divide-et-impera helps keep basic building blocks manageable
- Complex doesn't have to be complicated

Agenda

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^(1/2)

- 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

Finding the right package structure^(2/2)

- Group components and associated management modules in Python packages
- Use the **application and layer 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.)
- **Namespace objects** are one honking great idea, let's do more of those ... ☺

Data, classes and methods

- Use **methods** even for simple tasks...
 - but don't make them too simple
- Use **method groups** for more complex tasks / APIs
 - e.g. to implement a storage query interface

Data, classes and methods

- 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

- 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
 - regroup functionality
 - add more methods
- Magic word: **Refactoring**

Refactoring

- 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 and inline comments
 - doc-strings represent your method's contracts with the outside world
- Block logical units using empty lines...
 - Python loves whitespace ☺

Often forgotten: Documentation

- Document the intent of the methods, classes and logical code units...
 - not only their interface
 - and write tests as functional documentation
- Use descriptive identifier names...
 - even if they take longer to type

Quality Assurance: XP helps !

- Try to use some **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: More 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

Agenda

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

A photograph of a child wearing a red hat and a white shirt, sitting in a vast field of green grass and small white flowers. The child is positioned in the upper left quadrant of the frame, looking away from the camera. The field stretches out to the horizon, filled with numerous small white flowers, possibly daisies. The overall scene is peaceful and open.

Questions ?

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/>

Thank you for your time.

For more information, please write to info@egenix.com or
visit our website at <http://www.egenix.com/>

(c) Copyright 2013, eGenix.com GmbH